ERW

The Manual

Sebastiano Vigna

ERW: The Manual
by Sebastiano Vigna

Copyright © 2001, 2002, 2003, 2004, 2005 Sebastiano Vigna.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front–Cover Texts being “ERW”,“The Manual”,“Sebastiano Vigna” and with no Back–Cover Text. A copy of the license is included in the appendix entitled “GNU Free Documentation License”.

Table of Contents

Preface ..... **vii**
A Simple Example viii
1. Overall System Design **1**
Entity-Relationship Schemata 1
Cardinality Constraints 3
Weak Entities 5
Subtyping 5
Reification 6
Concurrency 7
2. ERL: an Entity-Relationship Language **9**
A Quick Tour 9
Types and Identifiers 9
The enum type 10
The fset type 10
Elements 10
erl 10
enumval 11
enum 11
fset 12
attr 12
isa 15
ent 15
leg 16
rel 17
3. The User Interface **19**
Stateless Editing 19
Simulating Remote Procedure Calls 20
State Discrepancies 20
Caching 21
4. Customisation **23**
User Interface Customisation 24
ERL Types vs. ERW Types 25
Attribute Specifiers 27
Custom Files 28
The Type Array 29
Customising Labels 29
Ordering Elements 33
Customising Filters 33
Hooks 35
The Lock Hook 36
The Pre-update Hook 36
The Post-update Hook 37
The Deletion Hooks 38
The Post-Authorisation Hook 38
The Selection-load Hook 39
The Button Hook 39
The Button Inhibition Hook 41
The Form-Loading Hook 41
Custom Forms 41
Form Resolution 42
ERF: an Entity-Relationship Form Language 42

5. Authentication and Authorisation **47**
Custom Authentication 47
The Authorisation System 47
Changing Owner 48
The Selection Level Tuning 49
Restrictions on Element-Based Authorisation 49
6. Transaction Support **51**
7. Installation and Configuration **53**
The Run-Time Environment 53
The `ERW-conf.php` Initialisation File 53
Configuring PHP 56
Configuring HTMLArea 57
Configuring FCKeditor 58
Configuring JSCalendar 58
8. `ERtool` **61**
Options 61
Checks 62
Error Messages 63
The DOT graphical back-end 63
The ERW Graphical Notation 64
Entity Types 65
Relationship Types 66
9. Standards and Browser Requirements **71**
10. Useful Functions **73**
Database Functions 73
JavaScript Functions 74
11. Localisation **75**
Language Configuration Variables 75
Other Localisation Variables 75
UTF-8 Support 75
12. Script Reference **77**
`ERW/checkERW.php` 77
`ERW/checkdb.php` 77
`ERW.php` 77
`list.php` 77
`form.php` 77
`default.php` 78
`chown.php` 78
`setprefs.php` 78
13. Troubleshooting **79**
`ERtool` problems 79
PHP problems 79
SQL problems 79
Configuration and Customisation Problems 79
When Nothing Seems to Work 79
Reporting Bugs 80
A. GNU Free Documentation License **81**
Preamble 81
Applicability and Definitions 81
Verbatim Copying 82
Copying in Quantity 82
Modifications 83
Combining Documents 84
Collections of Documents 84
Aggregation With Independent Works 84
Translation 85

Termination 85
Future Revisions of This Licence 85

Preface

ERW is a system for managing complex (and possibly large) databases using a web browser. Its original purpose was *content management*, that is, allowing easy maintenance of web site data organised in a database in a heterogeneous environment, where little is known about client machines except that they will have a web browser.

ERW is based on a formal description of the database in a variant of the common entity-relationship model (more precisely, of what it is usually called the *extended entity-relationship model*). An XML-based language, ERL (Entity-Relationship Language), is used to define formally a entity-relationship *schema* that provides a conceptual description of the database.

From the ERL-based description, a Java™ preprocessing tool, `ERtool`, *reifies* the schema, that is, it transforms the description based on entities and relationships in a set of tables that suitably implement that description. Moreover, it generates a graphical representation of the schema and documentation about the reification process; this documentation allows you to understand exactly how the process was carried out. And, last but not least, it generates a set of configuration files written in PHP. These files, when fed into the provided run-time PHP environment, generate a set of forms that allows easy editing of the database. A flexible customisation process allows you to tune the forms to your needs.

Where is the difference with other web-based database administration tools? The point is that ERW knows the abstract structure of the database, and thus can offer a much more sophisticated interface to the user. In particular, the user never sees an SQL table: rather, it is presented with relation-based operations such as "associate this element to this element". The run-time environment (using the configuration files produced starting from the ERL file) will modify the database tables correspondingly.

Just to make a comparison, a tool like `PHPmyAdmin` is to ERW as a disassembler is to a compiler/run-time environment. You can of course use `PHPmyAdmin` to do raw administration of the database tables generated by ERW, but the knowledge that ERW has of the abstract structure of the entity-relationship schema gives it the possibility of interpreting the same tables in a deeper way. Of course, ERW can only edit databases reified by `ERtool`, so it is less general.

These are not, however, the only advantages of using ERW. We list some of them:

- ERW has support for a wide range of features of the extended entity-relationship model, including multiple inheritance and multiple owners.
- ERW is entirely based on international standards and open-source tools. Since every part of the system is known and documented, you are never left with an unusable bunch of data, or outdated form-generating scripts that no one remembers how to modify. The system, moreover, is largely architecture independent.
- ERW is free software distributed under the GNU General Public License[1]. However, applications built using ERW may be distributed under the license of your choice (similarly to what happens to programs compiled with a free compiler).
- ERW maintains *referential* and *logical* integrity of your database. That is to say, not only editing cannot create dangling references, but also the cardinality constraints you have imposed will be automatically enforced; for instance, if every document must have an author, then it will be impossible to insert a new document without associating it with an author.
- ERW allows any number of users to access the database concurrently.
- ERW uses the W3C DOM to offer a rich and intuitive graphical interface. This interface comes at no cost once you have defined your entity-relationship schema in ERL.
- ERW supports gettext and UTF-8 everywhere: you can create forms and handle content in any language.

- ERW gives you for free user, group and element-based authorisation. Select, update, insert and delete privileges can be set up separately.
- ERW is based on a subset of SQL-99 that will work on almost every DBMS, and uses the PEAR[2] database abstraction layer, so you are not bound to a particular choice of DBMS.
- ERW uses ERL to define additional data types (such as enums) that are not available in SQL-99, and maps them to standard types.
- ERW provides *filesets*, that is, set of files with attributes that are related to an entity. This allows to easily add to the database multimedia data stored in the filesystem.

There are, of course, also some drawbacks. ERW makes a number of assumptions on the structure of the database and on the inner workings of your entity-relationship schema that prevents you from doing certain things:

- ERW is based on *local editing*: each entity has a corresponding editing form, and from that form you can edit that entity and adjacent relationships, but you cannot edit other components of your entity-relationship schema (you can, of course, from their respective forms).
- ERW uses a conservative approach in deletions: whenever an element is deleted, all incident relations are also deleted, unless this would break logical integrity, in which case the deletion cannot be performed. This is in contrast with the cascading approach, in which the deletion of an element can cause deletions of other elements so to maintain logical integrity. Cascade deletions would break the local editing paradigm, which is fundamental in ERW's design (there is an exception to this rules for weak entities, which are deleted together with any of their owners; see the Section called *Weak Entities* in Chapter 1).
- Presently ERW handles binary relationships only. If you really need relationships of greater arity, you will have to simulate them using fake weak entities with multiple owners.

A Simple Example

Maybe at this point you are curious about ERW, but you still do not understand what it exactly does. The easiest way to solve this problem is showing a simple example.

Suppose you have a library, and you would like to keep track of which users have (or had) a certain book. To this purpose you set up a simple entity-relationship schema, which has three entity types: `person`, `subscriber` and `book`. Book are lent to persons, but certain persons subscribed to the library, and possibly donated some money.

The schema contains a relationship type `loan` going from `person` to `book`, and this type has some *attributes*: first of all, the starting date `startdate` of the loan (this attribute is clearly mandatory, as there is no loan without a start); then, the ending date `enddate` of the loan (this attribute is optional, as the book could be currently out); finally, the attribute `type` that defines the type of the loan (long or short term).

The informal description of the entity-relationship schema that we have just given is formalised by the following ERL document:

```
<?xml version="1.0"?>

<!DOCTYPE erl PUBLIC "-//DSI//DTD ERL V1.2//EN" "file://erl.dtd">

<erl id="library" title="A Library">

&ERWauth;

<enum id="termlen" type="char" size="1">
 <enumval value="L" label="Long term"/>
 <enumval value="S" label="Short term"/>
```

```
</enum>

<ent id="person" label="Person">
 <attr id="fname" label="First Name" size="30" mand="true"/>
 <attr id="lname" label="Last Name" size="30" mand="true"/>
</ent>

<ent id="subscriber" label="Subscriber">
 <isa entref="person"/>
 <attr id="card" label="Card number" type="int" mand="true"/>
 <attr id="address" label="Address" mand="true"/>
 <attr id="donation" label="Donation" type="numeric" size="11" scale="2"/>
</ent>

<ent id="book" label="Book">
 <attr id="title" label="Title" mand="true"/>
 <attr id="author" label="Author" size="40" mand="true"/>
 <attr id="publisher" label="Publisher" mand="true"/>
 <attr id="isbn" label="ISBN Code"/>
 <attr id="year" label="Year" type="integer"/>
 <attr id="description" label="Description" type="text"/>
</ent>

<rel id="loan" label="Loan">
 <attr id="startdate" label="Start date" type="date" mand="true"
               default="'2001-01-01'"/>
 <attr id="enddate" label="End date" type="date"/>
 <attr id="duration" label="Type" type="enum" enumref="termlen" mand="true"/>
 <leg entref="person" label="Borrowed..."/>
 <leg entref="book" label="Lent to..."/>
</rel>

</erl>
```

There are several interesting features in this example. First of all, we include the system entity `&ERWauth;`, which will automatically include the entity and relationship types that are necessary to handle authorisation. Then we declare a *static enumerative type*, that is, a function from values to labels. The database will store the values, but the user will be always presented with the labels. Finally, we declare the entity types we are interested in: note that `subscriber` is declared as being a `person`.

Then, we declare our only relationship type. The two `leg` XML elements tell ERW which entity types must be related. Note also that we specify that the default value for the start date of a loan is the first day of the current millennium (but, should we be certain that our database fully supports the SQL standard, we could use, say, `CURRENT_DATE`).

It is now time for a little bit of magic: assuming there is an empty database named `test`, and that the file above is named `library.xml`, the command **`ertool --sql <library.xml`**will print on standard output the instructions to create your database. Usually piping this code in a DBMS client with the right privileges is enough to set up completely the database.

Now we have the database, but we still cannot modify it. Assuming that we installed correctly the ERW PHP scripts in a directory served by `Apache`, we create there a directory named `defs`, generate the definition files with **`ertool --defs <library.xml`**and move the PHP files thus produced to the `defs` directory. These files contain information used by the PHP run-time environment to produce the forms that handle the database.

Finally, you must set up a configuration file named `ERW-conf.php` that explains to ERW which kind of DBMS you are using, the name of the database and so on. There is a sample `ERW-conf.orig.php` file that you can copy and modify.

You now need a way to access the scripts handling form. The ERW scripts include a sample `index.orig.php` file that can be used as a start page (copying it to

`index.php`). If you do so, accessing with a browser the directory containing the ERW scripts you should see a list of links named as the entities we defined in the ERL file (the list is automatically generated for testing purpose only; the database administrator should set up a nicer way to access the editing forms). Clicking on them will lead you to the editing forms; for instance, to edit books you will get a list looking like this:

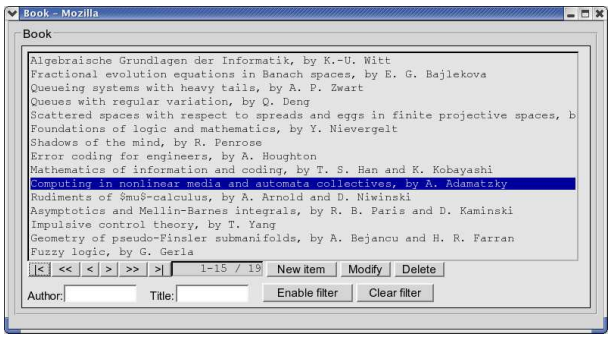
Book - Mozilla
Book
Algebraische Grundlagen der Informatik, by K.-U. Witt
Fractional evolution equations in Banach spaces, by E. G. Bajlekova
Queueing systems with heavy tails, by A. P. Zwart
Queues with regular variation, by Q. Deng
Scattered spaces with respect to spreads and eggs in finite projective spaces, b
Foundations of logic and mathematics, by Y. Nievergelt
Shadows of the mind, by R. Penrose
Error coding for engineers, by A. Houghton
Mathematics of information and coding, by T. S. Han and K. Kobayashi
Computing in nonlinear media and automata collectives, by A. Adamatzky
Rudiments of mu-calculus, by A. Arnold and D. Niwinski
Asymptotics and Mellin-Barnes integrals, by R. B. Paris and D. Kaminski
Impulsive control theory, by T. Yang
Geometry of pseudo-Finsler submanifolds, by A. Bejancu and H. R. Farran
Fuzzy logic, by G. Gerla
1-15 / 19
New item
Modify
Delete
Author:
Title:
Enable filter
Clear filter

Figure 1. An ERW list where you can choose books.

You can browse around, and filter elements (to be true, the filter shown has been customised). Once you find to the book you want to edit, you get to a form looking like this:

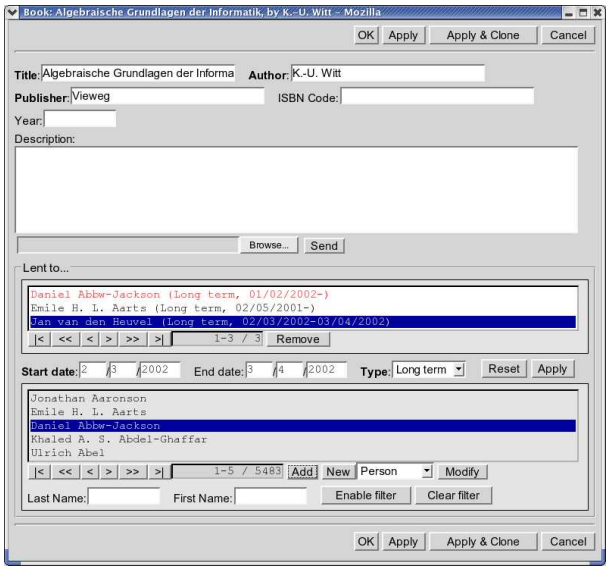

Figure 2. An ERW form for a book.

The first input elements allow you to edit the attributes of a book (note that you can upload a description text file). However, the rest of the form is concerned with editing a relation: there is a list of entities (persons that borrowed the book, in this case) and you can create and delete relationships, or change their attributes.

Analogous forms exists for persons, but here we have an interesting feature:

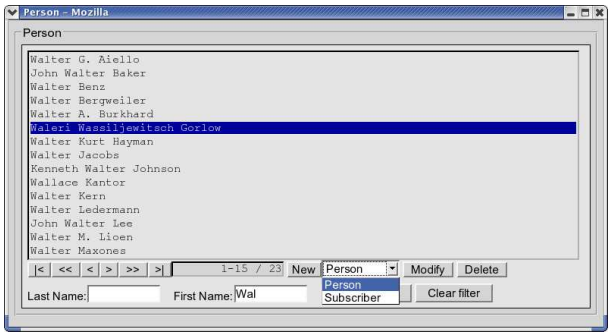

Figure 3. An ERW list where you can choose persons.

Note that you have the choice, when creating a new person, to make a subscriber or not. In general, you can select any of the subtypes of the current entity type. In any case, if you edit a person you will get the right form—a subscriber, for instance, will get something like this:

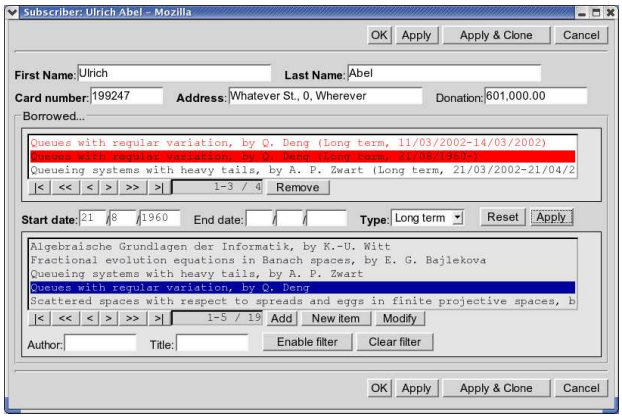

Figure 4. An ERW form for a subscriber.

You can also have a list of *subscribers only*: in this case, you can choose to have also cynical filters, such as the amount of donation:

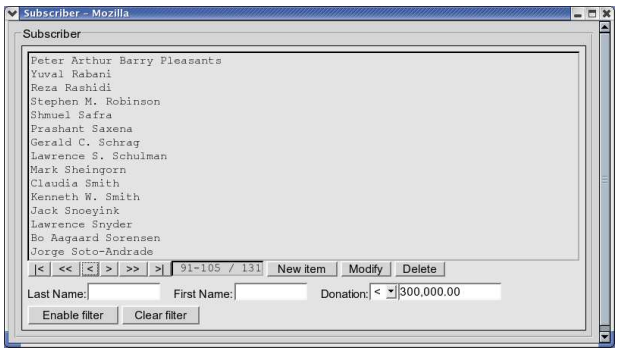

Figure 5. An ERW list where you can choose subscribers.

Finally, in certain cases you may want to edit *relationships directly*. That is, you may want to browse (with filtering) and modify loans by themselves:

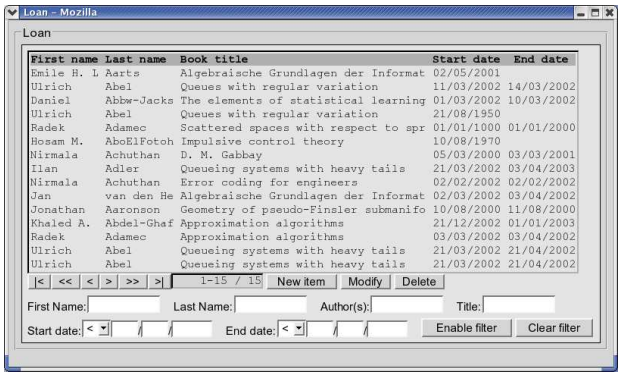
Loan - Mozilla
Loan
First name Last name Book title Start date End date
Emile H. L Aarts Algebraische Grundlagen der Informat 02/05/2001
Ulrich Abel Queues with regular variation 11/03/2002 14/03/2002
Daniel Abbw-Jacks The elements of statistical learning 01/03/2002 10/03/2002
Ulrich Abel Queues with regular variation 21/08/1950
Radek Adamec Scattered spaces with respect to spr 01/01/1000 01/01/2000
Hosam M. AboElFotoh Impulsive control theory 10/08/1970
Nirmala Achuthan D. M. Gabbay 05/03/2000 03/03/2001
Ilan Adler Queueing systems with heavy tails 21/03/2002 03/04/2003
Nirmala Achuthan Error coding for engineers 02/02/2002 02/02/2002
Jan van den He Algebraische Grundlagen der Informat 02/03/2002 03/04/2002
Jonathan Aaronson Geometry of pseudo-Finsler submanifo 10/08/2000 11/08/2000
Khaled A. Abdel-Ghaf Approximation algorithms 21/12/2002 01/01/2003
Radek Adamec Approximation algorithms 03/03/2002 03/04/2002
Ulrich Abel Queueing systems with heavy tails 21/03/2002 21/04/2002
Ulrich Abel Queueing systems with heavy tails 21/03/2002 21/04/2002
|< << < > >> >| 1-15 / 15 New item Modify Delete
First Name: Last Name: Author(s): Title:
Start date: End date: Enable filter Clear filter

Figure 6. An ERW list where you can choose loans.

The forms thus generated may not really suit your needs. In this case, you can try to customise them. A simple form of customisation, for instance, is localisation: labels and user interface elements can be translated in any language, and the correct language for the user will be negotiated with the browser. However, you can also define your own forms using ERF, an XHTML-like language; here we have a simple, self-explanatory example for a subscriber:

```
<?xml version="1.0"?>
<!DOCTYPE erf PUBLIC "-//DSI//DTD ERF V1.0//EN" "file://erf.dtd">
<erf ref="subscriber">
  <style type="text/css">
    fieldset { background: #B0B0B0; border: solid black }
    legend { background: #808080; border: black solid thin }
  </style>
  <fieldset>
    <legend style="background: #E0E0E0">User Data</legend>
    <input ref="person.fname" size="20"/>
    <input ref="person.lname" size="20"/>
    <input ref="address" size="80"/>
  </fieldset>

  <fieldset>
    <legend>Subscription Data</legend>
    <input ref="card" size="8"/>
    <input ref="donation"/>
  </fieldset>

  <p>You can see loan details by clicking on them.
  However, you are <em>not</em>
  allowed to modify loan data.</p>

  <list ref="person.loan_book" readonly="true"/>
</erf>
```

The resulting form would look as follows:

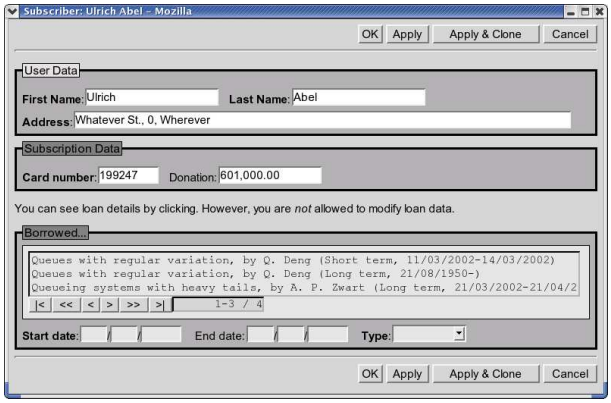

Figure 7. A customised form, with a read-only relationship display.

Essentially, we freely described a page where ERF input elements are replaced by ERW's input controls: besides changing the look of the page using style sheet, we can also set the size of the text inputs, force elements to be read only, and so on.

This example should have given a taste of ERW capabilities. Browsing this manual you can also find how to generate documentation and graphical layouts. To appreciate the power of conceptual modelling, you should think of schemata with hundreds of entities, complex type hierarchies and ownerships, which ERW will handle with the same ease.

Notes

1. http://www.gnu.org/copyleft/gpl.html
2. http://pear.php.net/

Chapter 1. Overall System Design

ERW is a complex system with many interrelated components, so before starting to describe in detail each part it is a good idea to review a bit more technically the procedure to generate and manage a database using it.

1. The first and most important step is the creation of an ERL file. ERL is a simple XML-based language that describes entity-relationship schemata decorated with additional information pertaining the actual database structure (such as the SQL types of the attributes) and some editing information (such as which relationships can be edited in an entity form). ERL has also support for simulated SQL types, that is, additional types that you can use as if they where actual SQL types. They include, for instance, boolean and enum.
2. Then, you have to create your database. This is accomplished using `ERtool`, a tool written in Java™ that reads ERL files and produces a number of outputs. The first usage of `ERtool` is to generate SQL code creating the actual database. Note that in this phase `ERtool` has to *reify* the entity-relationship schema, that is, to decide how to implement in the relational model the entity-relationship schema given in the ERL file (`ERtool` can also generate documentation on the reification process, so you can really know what it's doing).
3. Assuming that you installed ERW correctly (which includes configuring PHP and `Apache`), you now have to generate the definition files: these are PHP files that contains information about the database. They are essentially a reworking of the data contained in the original ERL file. The definition files are read by the PHP scripts of ERW's run-time environment to produce the actual forms.

   Definitions files are again generated by `ERtool` using the ERL file, and should be moved in a place where the run-time environment can find them (to this purpose there is a configuration variable that can be set).
4. Other configuration variables let you choose the type of DBMS, localisation features, and so on. Once everything is set up, your are ready to edit the database. You should provide a method to access the editing scripts with the right URL. A sample file, `index.orig.php`, can be used as a first try.
5. There are however several things that you could not like in the forms thus created. For instance, if you have a two-elements enumerative type (say, male or female) you could prefer a radio button to a drop-down menu. Or you could not like the way an element is labelled (by default, using its first attribute). For such things, ERW lets you write custom files, which alter some of the data of the definition files so to tune the user interface to your needs. Custom files also allow you to implant hooks in the run-time environment so that you can, for instance, do some post-processing after every database modification.[1]
6. If you need more flexibility in the user interface, custom forms can be used to provide a highly customised appearance. Custom forms are written in an XHTML-like language, so you can use style sheets, etc.
7. Finally, you may want to change the structure of your database. From the user interface viewpoint this can be done in a breeze, as it is just a matter of modifying the ERL file and regenerate the definition files. However, the changes to the database structure must be handled manually. Usually it is a good idea to keep a copy of the previously generated SQL statements and see the differences. In any case, simple changes such as adding or removing attributes, entities and relation are almost effortless, and the checkdb.php script can help you by pointing out discrepancies between definition files and SQL tables.

Entity-Relationship Schemata

To understand each other throughout the rest of this manual, we have to set up a minimal common language. Unfortunately, many terms in the database world are not completely standard, and can vary depending on the textbook you have. A radical choice made in this manual is to replace, whenever possible, database-related names for sets and functions with the more standard and more stable terms used by mathematicians. If you never heard of entity-relationship schemata (or of their graphical counterpart, *entity-relationship diagrams*), however, we suggest you to have first a look at a good textbook on the subject.

Entities model things of the real world. In our library example, a (specific) person and a (specific) book are both entities, because these are the things we want to model (we could call entities "things", but this would not sound very scholarly). Each entity has a *type*. Entity types model concepts: in our example, entity types will be `person`, `subscriber` and `book`. To an entity type we associate usually a set of *attributes*, that is, pieces of information: the book title, the person name, the subscriber donation and so on. Each entity defines a value for each attribute of its type (e.g. for each person there must be a name).

Relationships, on the other hand, model connections between things. For instance, a book may be on loan (or have been lent) to a person or not. Also relationships have a type, and may have attributes: for instance, the start and end date of the lent.

An *entity-relationship schema* is a directed[2] graph whose nodes are entity types, and whose arcs are relationship types. In other words, a schema is given by a set of entity types and a set of relationship types, where each relationship type is assigned a *source* entity type and a *target* entity type. In our example, the set of entity types is { `person`, `subscriber`, `book` }, whereas the set of relationship types is { `loan` }. The relationship type `loan` has source entity type `person` and target entity type `book`.

An *instance* of an entity-relationship schema is a sort of abstract database. It is given by a set for each entity type (called an *entity set*), and a *multirelation*[3] for each relationship type. Not surprisingly, elements of the set associated to an entity type are called entities, and elements of the multirelation associated to a relationship type are called relationships.

A multirelation is an extension of the mathematical concept of relation; in a multirelation from a set X to a set Y two elements can be in relation "more than once". Mathematically, a multirelation from the set X to the set Y is a set R (the *support* of the multirelation) equipped with two *legs* [4], that is, two functions R_0:R $\longrightarrow$ X and R_1: R $\longrightarrow$ Y. A multirelation should be interpreted as follows: if there is an element r of R such that R_0(r)=x and R_1(r)=y, then x and y are related (once). But if there is another element s of R such as R_0(s) and R_1(s)=y, then x and y are related twice, and so on. If there are no elements of X and Y that are related twice, then R is a standard relation. It will be useful in the following to note that given a multirelation R from X to Y, we can build a multirelation R^T, called the *transpose* of R, going from Y to X, simply exchanging the rÃ´le of R_0 and R_1.

Getting back to entity-relationship schema instances, we note that if a relationship type goes from an entity type instantiated in a set X to an entity type instantiated in a set Y, then the multirelation instantiating it must go from X to Y.

In our example, an instance of the database specifies a set of persons, a set of subscribers, a set of books and a multirelation for loans. Each loan (i.e., each element of the support of the multirelation) has an assigned person and book, and, of course, the same person can borrow the same book several times (so we really need a multirelation).

Finally, an instance of an entity-relationship schema specifies for each entity set and multirelation a function that assigns to each element its attributes. A precise definition of this function would require a precise definition of the attributes and of their Cartesian products, and it is beyond the scope of this section.

Getting it all together: entity-relationship schemata are a *syntax* that allows you to model some real-world objects and their connections. Each schema has, however, a *semantics*: its instances. The purpose of a database managed by ERW is to store an instance of an entity-relationship schema and let the user modify it.

> **Important:** The discussion above may have bored some reader, but it is necessary because ERW implements on the top of a standard SQL database the multirelational semantics of entity-relationship schemata described in "Multirelational Semantics for Extended Entity-Relationship Schemata With Applications", Proc. of ER 2002, Lecture Notes in Computer Science, Springer-Verlag, 2002. This semantics is much more precise and natural than the "tuple semantics" of relational databases, and in fact it is a good idea to think of relational databases in set-theoretic terms whenever you can.

Cardinality Constraints

An important feature of entity-relationship schemata is the possibility of specifying *cardinality constraints*. These are instructions of the form "every person has exactly one mother", or "every document must have at least an author (but possibly many)". There constraints are useful because they allow to maintain the *logical integrity* of the database: in a logically integral database, you can always trust a document to have an author, and you can avoid to check if it is missing. ERW maintains automatically the logical integrity of your database.

An important point (often missed) about logical integrity is that *it is a feature of the semantics of the entity-relationship schema, not of the underlying relational database*. It must be expressed abstractly in term of properties of multirelations, and not in term of the particular relational implementation of the entity-relationship schema.

ERW uses a slight extension of the standard way of giving cardinality constraints on relations (as it has to handle multirelations). Recall that each relationship type is an arc going from an entity type (the source) to another entity type (the target). Cardinality constraints restrain the possible multirelation that an instance can assign to the relationship type. For each of the two entity types you can specify a pair of indices between parentheses and separated by colon, as in `(1:N)`. The left character may be `0` or `1`, while the right character may be `1`, `N`, or `M`.

The specifiers assigned to a relationship provides restrictions on the cardinality (i.e., the number of elements) of certain relationships in an instance of the schema. More precisely:

- if the first character is `0`, it imposes no bound;
- if the first character is `1`, it forces each entity to be in relation with at least another entity;
- if the second character is `M`, it imposes no bound;
- if the second character is `N`, then a multirelation instantiating the relationship type must be a relation, that is, it cannot happen that entities are related twice;
- if the second character is `1`, it forces each entity of to be in relation with at most another entity (and it must be related once).

In other words, the first index is a lower bound on the cardinality of relationships involving an entity, while the second index is a sort of upper bound. There are some standard names: if the two pairs of indices are of the form `(`*x*`:1)`, then the relationship type is said to be *one-to-one*; if one is of the form `(`*x*`:1)` and the other of the form `(`*x*`:N)`, then the relationship type is said to be *one-to-many*; finally, if the two pairs are of the form `(`*x*`:N)`, then the relationship type is said to be *many-to-many*.

However, relations satisfying the restrictions above have standard mathematical names, which we will use in what follows. More precisely, we shall use the obvious extensions of these names to multirelations. A multirelation R from X to Y is said to be:

total

if for each element x in X there is at least one r in R such that $R_0(r)=x$;

monodrome (or a partial function)

if for each element x in X there is at most one r in R such that $R_0(r)=x$;

surjective

if for each element y of Y there is at least one element r of R such that $R_1(r)=y$.

injective

if for each element y of Y there is at most one element r of R such that $R_1(r)=y$;

a relation

if there are no distinct elements r and s of R such that $R_0(r)=R_0(s)$ and $R_1(r)=R_1(s)$.

Intuitively, a multirelation is total when every element of its source is related to something; it is monodrome if every element of its source is related (once) with at most one element of its target; it is injective if every element of its target is related (once) to at most one element of its source; it is surjective if every element of its target is related to something; and it is a relation if elements are never related twice.

These properties are strictly related: a multirelation is total if and only if its transpose is surjective, and it is monodrome if and only if its transpose is injective; finally, it is a relation if and only if its transpose is.

> **Note:** If you look back carefully, you will notice that the cardinality constraints of a relationship type associated to the source and target entity types are really expressed in terms of the legs of multirelations: indeed, cardinality constraints have nothing to do with the entity types; they are a purely set-theoretical feature of the multirelations instantiating a relationship type. One could equivalently say that a constraint of the form `(`*x*`:1)` forces the corresponding leg to be an injective function, and that a constraint of the form `(1:`*x*`)` forces the corresponding leg to be an surjective function; this definition is equivalent to the one above.

All in all, noting that a monodrome total relation is commonly called a *function*, we have the following table assigning standard mathematical term to the cardinality constraints specified on the source entity types (rows) and on the target entity types (columns):

	`(0:1)`	`(1:1)`	`(0:N)`	`(1:N)`	`(0:M)`	`(1:M)`
`(0:1)`	injective partial function	injective and surjective partial function	partial function	surjective partial function		
`(1:1)`	injective function	injective and surjective function	function	surjective function		

(0:N)	injective relation	injective and surjective relation	relation	surjective relation		
(1:N)	injective and total relation	injective and surjective total relation	total relation	total surjective relation		
(0:M)					multirelation	surjective multirelation
(1:M)					total multirelation	total surjective multirelation

The empty places represent impossible combinations. Note that if you exchange the two pairs of indices, you obtain the cardinality constraints on the *transpose* of the multirelation. Thus, if a multirelation has to be an injective relation, its transpose has to be a partial function, and viceversa.

Weak Entities

A *weak entity* is an entity that exists only if is related to a set of uniquely determined entities, which are called the *owners* of the weak entity. For instance, we could extend our library with a weak entity type `edition`; each book has several editions, and certainly it is nonsense to speak about an edition if this does not happen in the context of a specific book. From a user interface viewpoint, weak entities are usually edited in the context of (one of) their owners. When an entity is deleted from a schema instance, all owned weak entities are deleted, too. We shall call the type of a weak entity a *weak entity type*.

For entities of type `W` to be owned by entities of type `X`, a requirement must be satisfied: there must be an *identifying function* from `W` to `X` that specifies the owner of each entity of type `W`, that is, a relationship type going from `W` to `X` whose cardinality constraint impose its instances to be functions. Deletion of an entity of type `X` implies deletion of all related entities of `W`.

Weakness is recursive. If x owns y and y owns z, then x (indirectly) owns z. Of course, it must never happen that an entity is owned by another one by means of two distinct *ownership paths*; as a special case, it must not happen that an entity owns itself. `ERtool` performs a novel static check guaranteeing that this will not happen.

Subtyping

The last concept we need is that of *subtype*. When you say that entity type `A` is a subtype of `B`, or, more informally, that "`A` is a `B`", you are saying that all attributes of `B` are also attributes of `A`, and all relationship types incident to `B` are also incident on `A` (this phenomenon is usually called "subclassing" or "inheritance" in object-oriented jargon). In our example, `subscriber` is a subtype of `person`: indeed, each subscriber has a name, but has also other information.

ERW supports subtyping. For each entity type, you can specify, using the `isa` element, any number of supertypes, whose attributes and incident relationship types will be inherited (this is called *multiple inheritance* in object-oriented jargon). Obviously, you cannot form cycles (`ERtool` will prevent you from doing this).

Semantically, subtyping is modelled by set inclusion. That is, if *A* is a *B* then in every schema instance the set associated to *A* is a subset of the one associated to *B*. That is, every entity of type *A* is also an entity of type *B*.

To be completely precise, ERW adopts a restriction on the allowable assignments of sets to entity types: it is required that every entity has a definite type. For instance, if *A* and *B* are both a *C*, then it is not possible that an entity is in both *A*, *B* and *C* unless it is also in a common subtype of *A* and *B*. This avoids, for instance, that if `man` and `woman` are subtypes of `person` you can have an entity that is both a man and a woman (but should you need that, you just have to add a suitable subtype `hermaphrodite` of both `man` and `woman`).

Finally, ERW lets you mark an entity type *A* as *abstract*. In this case, there can be no entity whose type is *A*, but it is possible that there are entities whose type is a subtype of *A*. Of course, abstract types without subtypes are completely useless. But, should you need every person to be either a man or a woman (or possibly a hermaphrodite), you can declare `person` as abstract. Mathematically, the set of entities associated to an abstract entity type must coincide with the union of the sets associated to its proper subtypes.

Reification

Now that we have a complete abstract account of the semantics of a entity-relationship schema, it is time to *reify* it: this process maps our abstract multirelation-theoretic semantics to actual tables of a relational database.

ERW handles the reification process for you; in fact, it *has* to handle it, as the run-time environment will have to query and update an actual database, and there are several strategies to reify an entity-relationship schema. ERW follows a standard general-purpose algorithm that should work reasonably in all circumstances.

Reification of entity types is fairly obvious: each entity type is represented by an SQL table whose columns correspond to the attributes of the type (there is some other work to do to support ERW extended types, but we have not to concern ourselves with such details in this section), and whose rows represent entities. ERW adds an integer `id` primary key column to all tables, so to mimic the set-theoretic semantics we just discussed. Additionally, to support user/group based authorisation two additional columns `id_owner_usr` and `id_share_grp` might be added.

Even if the primary key is handled by ERW, you still have the possibility of defining set of columns that should not be duplicated using the `key` attribute of the `attr` element of ERL. The set of columns thus identified is declared `UNIQUE`. If you do not declare keys, you get a truly set-theoretical semantics: each entity is a distinguished object on its own, even if it has exactly the same attribute values of another entity (the same happens for relationships).

Reification of relationship types is more sophisticated: the obvious way is using a *support table* whose rows represent relationships (i.e., the elements of the multirelation) and the images of such elements under the two legs. In our library example, `ERtool` produces a table looking as follows:

```
CREATE TABLE loan (
   id int NOT NULL PRIMARY KEY,
   id0_person int NOT NULL,
   id1_book int NOT NULL,
   startdate date NOT NULL DEFAULT CURRENT_DATE,
   enddate date,
   duration char(1) NOT NULL
```

```
);
```

Again, the additional primary key `id` let us distinguish every relationship (even in multirelations), whereas `id0_person` and `id1_book` are the images of a relationship through the two legs. The remaining columns are attributes.

If, however, the cardinality constraints imposed force the multirelation instantiating a relationship type (or its transpose) to be monodrome (or even a function), there is a more efficient method: a column named `id_`*`relationship-type_other-entity-type`* is added to the table reifying an entity type. In this case, we say that the SQL table reifying the entity type *absorbed* the SQL table reifying the relationship type, or, informally, that the entity type absorbed the relationship type. Thus, for instance, if we modified our example and set `upper="1"` in the definition of the `book` leg (i.e., each book is on loan to at most one person), and moreover we eliminated all attributes from the relationship, instead of a support table we would have produced a new column `id_loan_person` in the table reifying the entity `book`. That is, `book` would have absorbed `loan`.

When `ERtool` reifies an entity-relationship schema, it first tries to absorb all relationship types without attributes into their sources (if it is possible); then it checks whether it would be possible to absorb the remaining relationship types without attributes into their targets, and complains if this is true, as by design absorption is possible only in the source entity types; finally, it creates a support table for all remaining types. There is some control on the first part of the process, as you can explicitly ask for a relationship type not to be absorbed even if it is possible.

Subtypes are realised by keeping in sync the `id` fields of the tables reifying the types involved. Thus, for instance, if *`A`* is a *`B`* then for each value of the `id` field in the table reifying *`A`* there is a corresponding row of *`B`* whose `id` field has the same value. This behaviour mimics exactly set-theoretical inclusion.

> **Note:** The reification process is more complicated, as, for instance, part of the reification of a dynamic enumerative attribute consist in creating an *ad hoc* table for storing the enumeration. The details of the reification process are given for each ERL element in detail in Chapter 2. In any case, `ERtool` generates detailed documentation about the reification process.

Concurrency

ERW is able to coordinate any number of users accessing concurrently a database. The design principle guiding the handling of concurrency is very simple, but effective: since editing is local, ERW keeps track of *all* changes made by the user in the browser. When the user submits its changes, ERW locks all tables involved with the editing of an entity before starting the update. Then, it checks whether referential or logical integrity might be violated by the modifications submitted by the user; then, it checks that read-only attribute values match the ones in the form; then, it perform custom checks using the pre-update hook (if any of the previous check fails, the transaction is aborted). Finally, it applies the required modifications, it performs custom updates using the post-update hook and unlocks the tables.

Locking is important because it guarantees atomicity. ERW does not assume fine-grained locking in the underlying database: rather, it locks all tables that could be possibly involved with an entity update or deletion. Unfortunately, an unpleasant side-effect is that in databases with a rich ownership/subtyping structure updating an entity may require locking a large number of tables. Note, however, that editing can be *always* carried out in parallel, because of ERW's stateless editing design. When changes are written to the database, however, the access will be suitably serialised.

Notes

1. In the future, custom files will be replaced by another XML-based language from which they will be generated.
2. *Directed* here means that arcs in the graph go from a node to another node, that is, they have a specified direction. Usually entity-relationship schemata defined without direction, but this is a big mistake, as relations are directed, and this can generate a certain amount of confusion (think of the relation "father of").
3. Traditional entity-relationship schema instances use relations instead of multirelations; however, multirelations are very useful in practise (as the library example shows), and they can be handled in a clear, formalised way using suitable mathematical definitions (cfr. "Multirelational Semantics for Extended Entity-Relationship Schemata With Applications", Proc. of ER 2002, Lecture Notes in Computer Science, Springer-Verlag, 2002).
4. The term "leg" comes from the category-theoretic usage of writing this kind of objects (called *spans*) with the R symbol in a raised position.

Chapter 2. ERL: an Entity-Relationship Language

ERL is an XML-based language that describes entity-relationship schemata enriched with information that is useful in reification and interface construction. It is the foundation over which ERW is built. To understand ERL, you must also understand how `ERtool` will reify ERL-defined entity-relationship schemata.

A Quick Tour

Before giving the reference to ERL elements, it is useful to have an informal discussion of what you can describe in ERL.

ERL documents are made of an `erl` element, that can contain `ent` elements (describing entity types) and `rel` elements (describing relationship types). Additionally, the `enum` element can be used to create *static enumerative types* (attributes of this type can assume a finite and fixed number of values), and the `fset` element can be used to create *fileset types* (fileset attributes associate to an entity a set of files with attributes).

Inside an `ent` element there are `attr` elements, describing attributes, and `isa` elements, describing supertypes. The `attr` element has some XML attributes[1] that let you specify the type of the attribute and whether it is mandatory (that is, whether the user must provide some input for that attribute).

Inside a `rel` element you can find, again, `attr` elements, but you will certainly find two `leg` elements, which specify the first and second leg of a multirelation (including the source entity, the target entity and cardinality constraints).

All elements let you insert optional text inside them; it will be added to the documentation.

Types and Identifiers

Almost all elements in an ERL document have a mandatory identifier. For the `erl`, `ent`, `rel` and `attr` elements, moreover, the identifier is bound to be *restricted*: it must start with an alphabetical character and must be entirely formed by alphanumerical characters. Case sensitivity cannot be guaranteed, as it depends on the underlying database, but ERW *is* case sensitive, so be careful. Note that the identifiers `id`, `file`, `owner`, `share` and `belongs` are reserved.

> **Warning**
>
> You can use identifiers of any length, but they will be combined during the reification process to get the actual name of database columns representing references to other tables. If you use too long identifiers, this construction could yield names that exceed the maximum identifier length in your database. Try to use short, meaningful identifiers.

All attributes have an ERL type. ERL types include the following SQL types:

CHARACTER	CHARACTER LARGE OBJECT
CHAR	CLOB
CHARACTER VARYING	NUMERIC
CHAR VARYING	DECIMAL
VARCHAR	DEC
NATIONAL CHARACTER	INTEGER
NCHAR	INT
NATIONAL CHAR	SMALLINT
NATIONAL CHARACTER VARYING	DATE
NCHAR VARYING	TIME

Moreover, there is an additional *simulated type*, BOOLEAN, which holds a truth

value[2], and TEXT, which is a non-SQL, but largely diffused, variant of CLOB. All SQL types are faithfully preserved during reification, whereas BOOLEAN is mapped to CHAR(1) (the column will hold `'0'` or `'1'`).

Finally, there are two *virtual types*: enum and fset. They were added to increase the expressive power of an entity-relationship schema described in ERL and avoid the creation of many relationship types with the sole purpose of simulating data types.

The enum type

Often in databases one has to store one out of a finite set of values. These values may be fixed forever (e.g., weekdays, male/female, ...), or could vary over time (e.g., positions in a company).

To this purpose, ERW provides two solutions by means of the enum type and of the `enum` element. The `enum` element specifies an SQL type and describes a list of value/label pairs. Any attribute of type enum pointing by means of the `enumref` attribute to a certain `enum` element will have the SQL type of that element; moreover, the content of the attribute is restricted to the values described in the `enum` element and, for each value, the user will always be presented with the corresponding label. In such a case we say that we defined a *static enumerative attribute*.

If, on the other hand, there is no `enumref` in an attribute with type enum, then a special table is created that holds the value/label pairs. The value/label pairs can be edited by authorised users, and so we speak of a *dynamic enumerative attribute*. Clearly this kind of enumerative attribute is more flexible, but also less efficient.

The fset type

Filesets are a rather sophisticated object (in fact, a weak entity type) camouflaged as an attribute. An attribute of type fset does not generate an SQL column: instead, it generates another table that holds information about files associated to entities. When editing an entity, the user will be able to upload and download files, associating them to the entity much like a weak entity.

As an additional facility, files in a fileset may have attributes (for instance language, or a title). To obtain this result, a fileset type defined using the `fset` element must be referred using the `fsetref` attribute. Of course you cannot use the fset type when defining the attributes of a file set.

Elements

In this section we list all ERL elements, giving in detail their usage and meaning.

erl

Name

`erl` — A whole entity-relationship schema

Synopsis

The `erl` element is the top-level element of an ERL document. It contains the `ent`, `rel`, `enum` and `fset` elements.

Attributes

id

A restricted identifier for the whole schema; the definition file carrying global information about the database structure will be named after this attribute.

title

An optional descriptive title for the whole schema.

enumval

Name

`enumval` — A value/label pair of a static enumerative type

Synopsis

The `enumval` element describes a value/label pair for a static enumerative type. The user will always see the label, but the database will store the corresponding value.

Attributes

value

The value to store in the database.

label

The label to present the user with.

enum

Name

`enum` — A declaration of a static enumerative type

Synopsis

The `enum` element contains a list of `enumval` elements describing the value/label pairs of a static enumerative type.

Attributes

id

An XML identifier for this type.

label

A label for this type, mainly for documentation purposes.

type

An SQL type for this attribute. If not specified, it is assumed to be INT.

size

For the types for which it is meaningful, the size of the attribute. See attr.

scale

For numeric types, the scale, that is, the number of fractional digits. See attr.

fset

Name

`fset` — A declaration of a fileset type

Synopsis

A fileset type specifies which attributes each file of the fileset will have. The `fset` element contains a list of `attr` elements to this purpose.

Attributes

id

An XML identifier for this type.

label

A label for this type, mainly for documentation purposes.

Reification

A fileset type declaration does not *per se* generate any SQL code. However, it defines the content of the support tables for all attributes of type fset that refer to the type, as all attributes listed inside the element are reified and added to the table.

attr

Name

`attr` — An attribute of an entity or relationship type

Synopsis

The `attr` element describes an attribute of an entity or relationship type.

Attributes

id

A restricted identifier for this attribute. Note that the identifier is not an XML identifier, so you can use the same attribute identifier for attributes of different types.

label

A label for this attribute. The user will be always presented with this label when dealing with the attribute.

help

A detailed help text for this attribute (for tooltips).

type

An ERL type for this attribute. If not specified, it is assumed to be VARCHAR.

enumref

If the `type` attribute is set to `enum` and you want to specify a static enumerative type, you should specify here the XML identifier of an `enum` element.

fsetref

If the `type` attribute is set to `fset` and you want the generated fileset to have attributes, you should specify here the XML identifier of an `fset` element.

size

For the types for which it is meaningful, the size of the attribute. For text types, it is the maximum number of characters that can be stored, and defaults to 255. For numeric types, it is the number of meaningful digits.

scale

For numeric types, the scale, that is, the number of fractional digits (by default, 0). Note that the scale is included in the size, so a numeric type with size *n* and scale *s* has *n*-*s* digits to the left of the point and *s* digits to its right.

regex

A JavaScript regular expression defining the valid content of this attribute. This is meaningful only for text and numbers, and provides a generic, flexible way of checking user inputs. The user will be prevented from setting the content of this attribute to a string that does not conform to the regular expression. Note that the empty string is always a valid input for a non-mandatory attribute.

> **Warning**
>
> The input provided by the user is valid if it matches somewhere the regular expression; if you want to give a tight constraint (i.e., on the whole content), you should start your expression with ^ and end it with $, as usual. Note that, to avoid to confuse the user, it is probably a good idea to permit trailing spaces.

default

A default value for this attribute. The allowed content depends on your database, although by SQL-99 standard you can use a literal, `CURRENT_DATE`, `CURRENT_TIME` and a few other specifiers (in PostgreSQL you can use any variable-free SQL expression). For example, by setting `default` to `CURRENT_DATE` for a date the user will be presented with the current date as default value.

> **Warning**
>
> ERW can perform little or no check on the content of this attribute. Try to use standard SQL constructs for portability, and check carefully the resulting type. In particular, constant strings must be quoted (e.g., **`default="'a string'"`**).

mand

This field can be set to `true` or `false` (the default). In the first case, the user is required to insert some data for this attribute.

trim

This field is meaningful only for ERL text types, and can be set to `true` (the default) or `false`. In the first case, this attribute will be trimmed. More precisely, trimming happens following the ERW type associated (see the Section called *ERL Types vs. ERW Types* in Chapter 4): for type `pw`, no trimming; for type `t`, left and right trimming; for type `a`, left trimming.

key

This field can be set to `true` or `false` (the default). The set of fields for which this attribute is true constitutes the *key set* of the entity or fileset. ERW will prevent you from inserting two elements with the same key set. Note that for weak entity types the leg of the identifying functions are part of the key (see rel).

hidden

This field can be set to `true` or `false` (the default). In the first case, the user is never presented with the attribute, but the attribute is retrieved and rewritten without change on updates. This allows to create fields to store additional information not edited through ERW (but maybe computed each time the element is updated).

Reification

An attribute that is not of type enum or fset is reified by an SQL column in the obvious way. Enumerative types of identifier *`attribute-name`* generate a column named *`attribute-name`* in the case of a static enumerative attribute (and the type of the column will be given by the static enumerative type), whereas they will generate a column named `id_`*`attribute-name`* of SQL type INT in the case of a dynamic enumerative attribute. This column points to a table named *`entity-type_attribute-name`* that contains the actual elements of the type. The table can be edited much like an entity table.

As we mentioned previously, attributes of type fset do not generate an SQL column, but rather a support table (again, named *`entity-type_attribute-name`*) containing the attributes of the fileset, if a file set type is present, and two additional columns: `file`,

holding the actual name of a file in the file system, and `id_belongs_`*`entity-type`*. Each file stored in the fileset points to the entity it belongs to using the latter column.

isa

Name

`isa` — A reference to a supertype.

Synopsis

An entity type can inherit attributes and relationship types from any number of supertypes. The `isa` element specifies a supertype of an entity type.

Attributes

entref

The identifier of the entity this element points to.

Reification

When you specify that *A* is a *B* ERW keeps in sync the tables reifying the two types (i.e., for each entity in *A* there is a corresponding entity with the same id in *B*). Whenever you edit an entity of type *A*, independently of whether you are editing it in the context of *A* or *B*, ERW knows that it is of type *A* and acts accordingly.

ent

Name

`ent` — An entity of an entity-relationship schema

Synopsis

An entity type models a concept by means of attributes. The `ent` element describes an entity type of an entity-relationship schema. It can contain text that describe informally its use or meaning, and any number of `isa` and `attr` elements.

Attributes

id

A restricted identifier for this entity type.

label

A label for this entity type. The user will be always presented with this label when dealing with entities of this type.

abstract

This field can be set to `true` or `false` (the default). In the first case, the entity type is declared to be abstract (see the Section called *Subtyping* in Chapter 1).

owner

This field can be set to `true` or `false` (the default). In the first case, for every entity of this type there is a user who holds all privileges about the element (i.e., you have element-based authorisation); moreover, each element can be assign to a group, and then all users belonging to the group hold all privileges about the element (the amount of privileges, however, can be controlled; see the Section called *The Authorisation System* in Chapter 5). There are some restrictions on the entity types for which this attribute can be set to `true`—see the Section called *Restrictions on Element-Based Authorisation* in Chapter 5.

Reification

An entity type is always reified by an SQL table named as the XML attribute `id` and having as columns the reification of its attributes. Note however that because of virtual types the reification of an entity type may in fact create several SQL tables, namely those for dynamic enumerative attributes and those for filesets. Additional columns may also appear in the reification of an entity type if relationship types have been absorbed into it.

Note that when you require element-based authorisation you are also requiring it for all subtypes. For this reason, to avoid inconsistencies only certain entity types can have element-based authorisation. See the Section called *Restrictions on Element-Based Authorisation* in Chapter 5.

leg

Name

`leg` — A leg of a relationship type in an entity-relationship schema

Synopsis

The `leg` element describes one of the legs of a relationship type.[1]

Description

A leg is given by its codomain (an entity type) and information on the cardinality and editing constraints.

Attributes

entref

The identifier of the entity type this leg points to.

label

A label for this leg. The user will be always presented with this label when dealing with the relationship type this leg appears in *from the pointed entity type*.

help

A detailed help text for this leg (for tooltips).

lower

The lower bound for this leg. It may be `0` (the default) or `1`.

upper

The upper bound for this leg. It may be `1`, `N` or `M` (the default).

key

This field can be set to `true` or `false` (the default). In the first case, the entity type referenced by this leg becomes weak, and the entity type referenced by the other leg becomes one of its owners. The cardinality constraint on this leg must be `(1:1)`, and the one on the other leg (*x*`:N`).

editable

This field can be set to `true` (the default) or `false`. In the first case, the relationship type containing this leg will be editable from the entities of the pointed type.

Reification

See rel.

Notes

1. Technically we defined legs on multirelations, which are the semantics of relationship types, but the reader will forgive us if we extend the term.

rel

Name

`rel` — A relationship type in an entity-relationship schema

Synopsis

A relationship models a connection of some kind between entities. The `rel` element describes a relationship type. It must contain two `leg` elements and any number of `attr` elements. The first leg is taken to be the source of the relationship type, and the second leg its target.

Attributes

id

A restricted identifier for this relationship type.

label

A label for this relationship type. The user will be always presented this label when dealing with the relationship type abstractly (i.e., without starting from an entity type).

absorbable

This field can be set to `true` (the default) or `false`. In the second case, `ERtool` will always generate a support table to reify the type.

Reification

The reification techniques used by `ERtool` have been described in the Section called *Reification* in Chapter 1. Here we dwell into the details of field naming and so on.

First of all, not all cardinality constraints are supported by ERW: if on both legs the upper bound is `1`, then at least one leg must have a lower bound `0`. Bijections, which are discarded by this restriction, are useless (as they make it impossible to modify locally the database).

A nonabsorbed relation with source and target entities *`A`* and *`B`*, respectively, gets two additional columns `id0_`*`A`* and `id1_`*`B`*, that are used to point at the elements of the two entities by means of the `id` column; in other words, they give the values of the two legs on the elements represented by a SQL row. Note the index after `id` it is necessary, as ERW support loops (relationships from an entity to itself).

On the contrary, a relationship *`R`* from entity *`A`* to entity *`B`* giving rise to a partial function is reified as an additional column named `id_`*`R`*`_`*`B`* of SQL type int in *`A`*. Injective relations (whose transpose would be absorbable) give rise to an error. You just have to exchange the rÃ´le of the two leg specifiers.

Notes

1. Unfortunately, the word “attributes” has different meanings in the XML and entity-relationship world.
2. This is actually an SQL-99 type, but it is not supported by many DBMS

1. Technically we defined legs on multirelations, which are the semantics of relationship types, but the reader will forgive us if we extend the term.

Chapter 3. The User Interface

This chapter describes the basic mechanisms and features of the web user interface generated by ERW. It is not an end-user manual.

As we mentioned before, ERW lets you edit instances of an entity-relationship schema. That is, it lets you create and delete entities and relationships, and change their attributes.

Each entity has an associated form, from which the user can modify its attributes and incident relationships. To access the form, the user must select the entity from the *main list*: it is a list of all existing entities. This list can be filtered in several ways, and is generated by the script `list.php`.

Once the user selects an entity for editing, a new window is opened that contains HTML input controls for attributes (text inputs, text areas, radio buttons etc.) and a number of complex structures for editing relationships. The content of the window is generated by the script `form.php`.

The rationale behind the editing of relationships is that you have a white-background list, containing the current relationships; and, immediately below, a grey-background list of entities that could be used to create new ones. Suitable buttons allow the user to perform relationship editing. For convenience, buttons that allow to create new entities or modifying existing ones are also allowed, but they can be inhibited.

Editing relationship attributes is also fairly easy: there is a set of input controls under the list of relationships. All new relationships get the attributes currently displayed. Moreover, whenever the user clicks on an existing relationship, its attributes are loaded into the controls (in this way it is easy to copy the attributes of an existing relationship). If the user wants to modify the attributes of an existing relationship, he or she has just to click on the relationship, modify the controls and click on the "Apply" button, which will apply the current values.

You can help the user to understand what's going on my making good use of the `help` attribute for `attr` and `leg` elements.

By default, ERW displays first all attributes, then filesets, and then relationships. Attributes are listed in the same order as in the ERL file; they are prefixed with their label, and the label is embedded in a `STRONG` HTML element if the attribute is mandatory. Of course, this can be changed using custom forms.

If an attribute cannot be modified, because it is read only, it is displayed in a different manner: HTML input control get their `READONLY` attribute set, whereas lists just display the available elements (e.g., relationships) on a grey background.

> **Warning**
>
> Often browsers do not support the `READONLY` attribute on sophisticated controls, such as drop-down lists. Of course, ERW will just discard any changes, but the user may not be aware of this fact.

Finally, the user can select among four buttons (any of which can be inhibited) to finish editing. "OK", as expected, save changes and closes the form. "Save" commits changes, but leaves the form open for further editing. "Save & Clone" commits changes, and creates a clone of the current entity that can be further edited and saved (this is most useful when doing repetitive data entry). Finally, "Cancel" will forget all changes.

You can tweak the `default.php` script, which generates the CSS style sheet, to change some details. In particular, `DIV` elements of class `listborder` surround lists: by choosing, for instance, the value `scroll` instead of `hidden` for the `overflow` CSS attribute you can display lists with scroll bars that let the user scroll to see long labels. Moreover, the `editable` attribute of the `leg` element allows you to select from which side the user can edit relationships.

Stateless Editing

The main ingredient in the design of ERW's web interface is *stateless editing*. HTTP is a stateless protocol, and, as such, is not prone to support stateful interaction with a server (this is usually achieved with a mix of cookies and server sessions imposed over the protocol). ERW is designed to let the user edit freely a database recording *no* information server-side. All state related to the modification made by the user is stored client-side, so that the connection to the database is truly stateless.

This approach has a number of advantages: in particular, it adheres to the *principle of least surprise*: web users are by now accustomed to the idea that closing abruptly a web page (or even killing the browser), even in the middle of a multi-page form submission, will result in no changes being recorded server-side.

Indeed, any web form is an example of stateless editing: input controls store client-side the current choice of the user. The situation, however, is completely different when we take into consideration relationship editing.

The user, before submitting any information to the server, should be able to add relationships, delete old or new ones, and edit their attributes, *without storing state on the server*. This requires a completely different approach, as the state of input controls is not sufficient to store this information.

ERW uses JavaScript scripting and the W3C DOM to alter the appearance of the form, showing how relationships are added or deleted, and how their attributes are modified. *All these data are stored in the JavaScript state of the browser*. Suitable JavaScript code interacts with the database presenting the user with a modified view, which however does not really exist server-side. When the user has finished with editing, and submits the form, the entire JavaScript state of the form is suitably serialised and sent to the server. The PHP run-time environment on the server locks the part of the database that is affected by the changes, checks that no integrity constraints are violated, and performs the changes. If anything goes wrong, the user is presented again with the form that was submitted: using the serialised client state, the server is indeed able to completely rebuild the JavaScript state of the submitted form.

Simulating Remote Procedure Calls

JavaScript provides no direct way to connect to a database. Usually, whenever the user has to choose among a set of elements the entire set is packed in a suitable HTML input element (e.g., `SELECT`) and sent to the browser.

This approach, however, makes it impossible to edit large databases. ERW aims at the greatest generality, and thus implements on top of HTTP a simulated remote procedure call (RPC) mechanism using a hidden frame (this technique is often called *remote scripting*).

Every form contains an invisible `IFRAME` element that is used to interact with the server. Whenever the user interface code needs some data from the database (for instance, because the user is browsing a set of entities) the location of the `IFRAME` element is set to a particular URL that will load into the `IFRAME` element some JavaScript code; in turn, this code manipulates via the DOM the user interface, shows these data to the user, and updates the JavaScript state. The effect is much like an RPC.

State Discrepancies

Statelessness has a relevant consequence, which users can find disorienting at first: the state of the browser is not necessarily synchronised with the content of the database. For instance, if someone else adds an entity, it will not show up in *your* main or selection lists, at least until you re-filter their content, or move around with arrows.

Correspondingly, it is possible that you deleted, say, a relationship, without breaking any cardinality constraint: however, since in the mean time another user has deleted other relationships, your operation becomes illegal. In this case ERW will warn you, and show you again the form you were editing (this time, with updated information about the database content).

Caching

A page with a rich set of features (text areas, filesets, etc.) can fire, in principle, several dozens of parallel HTTP connections, as not only the main page, but also all relevant `IFRAME`s must be filled. Each script in turn includes JavaScript code, style sheets and so on.

To lessen the burden on the HTTP server, ERW tries to make cacheable for six hours all URLs that do not contain database-dependent data. This include most of the JavaScript and CSS material. In this way, the load is essentially reduced to one URL per `IFRAME`.

A possibly pernicious effect is that when you make an upgrade there could be a mix of old and new code all running in the same browser. Moreover, to make things work you must be sure that clocks are set properly.

This kind of problems are however easily solved by forcing the reload of the page involved, or, for example, by upgrading during an enough long period of system inactivity.

Chapter 4. Customisation

A serious problem faced by automatic software generation for content management is the existence of two contradictory goals: the application should be generated in a completely automatic way, to minimise efforts, but at the same time should be highly customisable, to satisfy the needs of the users.

Often the solution is adding custom code to an automatically generated skeleton: the custom code can then take care of the details of the implementation that would be difficult or impossible to express. However, even if this approach is extremely flexible (as essentially anything can be implemented on top of the skeleton, if sufficient documentation is available) it is also very dangerous. Usually, procedurally specified customisations end up being strongly tied to a specification: at that point, deep modifications to the specification lead to unpredictable behaviour.

ERW adopt a design pattern that we term *specification percolation*: the generation of the user interface is akin a fluid, percolating from the ERL specification to the user interface through a series of additional specifications, which act similarly to active membranes: they may filter some information, and even replace it with something else, but it should (almost) never happen that information is lost in the process or an inconsistent configuration is generated.

More down-to-earth, customisation of ERW is a process by which you can alter certain standard choices made by `ERtool` when producing the definition files, create custom forms, and add external code to the ERW standard database processing code so to perform special actions.

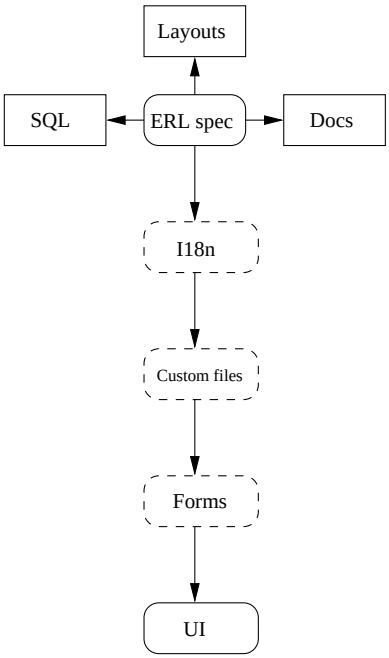

Figure 4-1. Specification percolation.

In Figure 4-1 we represent schematically the way information percolates through ERW's additional specifications.

The first membrane on the route of this information to the user interface is localisation: using the standard GNU gettext package, labels for attributes and enumerative type values are translated in a suitable language (usually negotiated with the browser). Localisation is discussed in Chapter 11.

Then, the administrator may set up *custom files* which alter (in a controlled way) the information represented in the definition files. This modifications may range from filtering options (specifying which fields are important to find quickly entities and relationship of a given type), labelling (`printf`-like formatting strings that use the attributes of the schema), and alternate user interfaces.

This information allows to tune several parts of the application, but it is not necessary to specify any part of it: for instance, filtering is performed by default on the first mandatory attribute, and labelling is handled analogously.

Then, the user may specify custom forms. These are completely redefined forms, which are described in ERF, an XHTML-like language that contains the basic elements, plus suitable elements to position the input controls for attributes and relations. Of course, forms must satisfy some constraints (for instance, they must contain all mandatory attributes), but this is checked automatically.

Again, specifying a custom form is not necessary: ERW is able to generate autonomously a default form, using customised filtering options, alternate user interface and so on. Of course, these may be all re-specified at the custom form level, but this is not necessary: a custom form may decide, for instance, to modify just the *order* in which attribute are presented, but not the specific interface that was chosen for each of them.

The advantage of a careful design based on specification percolation is that the application is highly customisable, but at the same time *every change to one of the specifications is immediately reflected in the user interface*, as it percolates freely through the following ones.

To clarify the usefulness of this approach, we state a few examples:

- In a complex database, with a high degree of customisation and localisation, the administrator has to add quickly a new entity type. This is as simple as adding the new type to the ERL specification: there is no immediate need to add all customisation layers to the new type, as it is guaranteed that the information contained in the ERL file will percolate down to the user interface, producing a reasonable form in the default language. Layers of customisation can be added later, without disturbing the rest of the application.
- After defining a large number of custom forms, we realise that a certain entity type will contain always a small number of entities. Thus, we wish that all many-to-one relations pointing at that entity type are displayed as `SELECT` elements (i.e., a drop-down list), instead of using the standard display (which is much larger). In this case, we just have to set up our choice during the customisation phase, so that it will percolate correctly to the forms (even if they are customised).
- We decide that we need an additional attribute in an existing entity type. Of course, the new attribute will immediately show up in the default form. However, if there are custom forms for that entity type, they are likely not to contain the new attribute. ERW detects this situation and adds automatically the attribute as *hidden*—it will be silently preserved when editing the database, and filled with its default value. The administrator will be warned that it is the case to modify the custom forms accordingly, but again consistency will be maintained.

User Interface Customisation

The user interface provided by ERW can be customised in several ways, but the most important point is that *the same data can be presented in different ways*. By default `ERtool` will use an interface that will work in every case; however, it could not be the best interface for a particular case. For instance, if a relationship is absorbed, ERW present it as a complex set of user inputs where the user can filter and choose which element to associate to the current one. However, if the choice is among a very small number of elements, a drop-down list could be a better choice.

In this section we will first review some general idea about customisation: then, we will see how to write custom files, and finally custom forms.

ERL Types vs. ERW Types

When you create an ERL file, you specify an ERL type for each attribute. This type is automatically mapped by `ERtool` on an ERW type, which specifies the user interface for that attribute. Moreover, an ERW type is assigned to each relationship type incident to an entity type.

In the following table you can find the correspondence between ERL types, relationships and ERW types; the types that `ERtool` assigns by default are marked with an asterisk:

CHARACTER CHAR CHARACTER VARYING CHAR VARYING VARCHAR NATIONAL CHARACTER NCHAR NATIONAL CHAR NATIONAL CHARACTER VARYING NCHAR VARYING	t* a fa ha pw
CHARACTER LARGE OBJECT CLOB TEXT	a* fa ha t pw
TEXT	a*
INTEGER INT SMALLINT	i* p
NUMERIC DECIMAL DEC	n* ns
ENUM (dynamic)	e
ENUM (static)	s* r
BOOLEAN	b
DATE	d* jd
TIME	h
FSET	f
absorbed relationship types	r1* r1s
transpose of absorbed relationship types	r1t*
transpose of identification functions (for weak entities)	r1w* r1e
non-absorbed relationship types	rN

t

The most common type, a string of text. The user is prevented from inserting more characters than the SQL field can store.

pw

As above, but the input field is obscured (the user just sees asterisks while typing). Useful for passwords.

a fa ha

A fairly large amount of text (currently limited to 64KiB) displayed in an HTML `TEXTAREA` element. The user has the possibility to upload a file into the text area. If you choose the ha type, then on browsers supporting HTMLArea you will get a

WYSYWYG HTML editor (of course, HTMLArea must be properly configured). Analogously for fa and FCKeditor.

i p

The second most common type, an integer. There is no check on the actual dimension of the number inserted by the user, as it could depend on the DBMS. The type p forces its content to be (strictly!) positive.

n ns

The exact numeric types. These types require a size and a scale in their declaration, so ERW can check that the user is not inserting more digits than allows. The type ns presents the same data in separated and localised form, that is, the user can use a localised comma and a localised point, and, moreover, the numbers inserted will be automatically thousand-separated (if they are not already) as the user leaves the input field.

e

A dynamic enumerative type. The user is presented with a drop-down list.

s r

A static enumerative type. The user is presented with a drop-down list with type s, and with a radio button set with type r. Note that it is not a good idea to set type r for a non-mandatory attribute, as the user has usually no way to deselect all buttons.

b

The boolean type. The user is presented with a checkbox.

d jd

The date type. The user is presented with three text inputs for day, month and year. If you choose the jd type, then an additional button opening a pop-up JSCalendar will be added (of course, JSCalendar must be properly configured). Note that if you choose jd and JSCalendar is not present the button will not appear, but no errors or warnings will be generated.

h

The hour/minutes type. The user is presented with two text inputs for hour and minutes.

f

The fileset type. The user with a set of input elements that allow one to upload files to the server, possibly setting or editing their attributes.

r1 r1s

The type of an absorbed relationship type. By default the user is presented with a set of input elements that allow one to select an entity to form a relationship. The type r1s, instead, simply creates a drop-down list containing all entities among which to choose, but clearly this is not a good idea if the entities are many.

r1t

The type of the transpose of an absorbed relationship type. By default the user is presented with a set of input elements that allow one to select an entity to form one or more relationships.

r1w r1e

The type of the transpose of identification functions. The types r1w and r1e are thought for weak entities, and they lets you create new entities that will be automatically owned by the current entity.

There is however an important difference: for the r1e type, the editing is *embedded*: the attributes of the other entity are embedded into the form of the main entity, and the user can modify and edit them much in the same way a relationship with attributes can be edited. Embedded entities will be created or deleted following the user editing. This approach has a major limit: you can only edit attributes of the embedded entity, and not relationships. Thus, in particular, no total relationship can go out of the embedded entity.

On the contrary, using the r1w type one gets a new form to fill for a new weak entity. The form can modify, of course, its relationships. However, the new element is actually created and associated to the owner. Note that because of this difference, it is not possible to associate a new weak entity to a new entity (i.e., to an entity that has not been written to the database). The user will be requested to save the current changes to the database to continue.

rN

The type of a non-absorbed relationship type. The user is presented with a set of input elements that allow one to create a relationship and modify its attributes.

A particular case, to handle with care, is that of *loops*. A relationship type can have the same source and target (e.g., "father of"), and in this case for absorbed relationship types you must choose *one* direction for editing; non-absorbed relationship types can only be edited forward.

There is, however, a special important case: when you define a relationship type between entity types with a common subtype, it is treated as a loop (as indeed you can associate entities of the common subtype to entities of the same type). In this case, non-absorbed relationships can be edited only from *the type that is highest in the type hierarchy* (where *highest* means appearing later in a breadth-first visit of the hierarchy). This has a side effect: if the relation is total in the opposite direction, you will most probably get errors. This configuration however is purely theoretical and indeed it is so contrived that it should never happen in real schemata.

Attribute Specifiers

Customisation requires specifying attributes, possibly of related types (e.g., for filtering, displaying, ordering, etc.). An *attribute specifier* lets you access dynamically attributes and relationships of supertypes, subtypes and related entities when filtering, labelling or ordering. However, you must understand completely the pitfalls involved, in particular the cases when an attribute specifier may have a `NULL` value.

The general form of an attribute specifier is

`[`*`leg-codomain`*`.]{`*`relationship-type`*`->}[`*`qualifier`*`.]`*`attribute`*

The formal description above is quite unfathomable. However, it embodies a simple idea: you start from the current type and move along relationship types, reaching the type containing the attribute you are interested it. At the end, you may need to jump to a supertype or a subtype to get the attribute you need: this is the rÃ´le of *`qualifier`* (note that jumping to a supertype guarantees the existence of the attribute, but jumping to a subtype could lead to a non-existing attribute, which would be valued `NULL`). Moreover, if you start from a relationship type, you may want to move to its source or target entity type: this is the rÃ´le of *`leg-codomain`*, which must be of the form *`entity-type_leg-index`* (for an example, see the Section called *Displaying Relationships*).

Thus, for instance, you can access the name of a subscriber with `person.name` (here `person` is a *`qualifier`*), or the name of the book a loan refers to with `book_1.title` (here `book_1` is a *`leg-codomain`*). Of course, specifying a *`leg-codomain`* when the starting type is not a relationship type will give rise to an error. There is also an attribute `id` that exists in every type and it is guaranteed to provide a positive distinct number for every element of a given type.

Following Identification Functions

More interestingly, we can access attributes of related entities (e.g., owner entities) using the

```
relationship-type->attribute
```

notation (the relationship type must be named as an ERW attribute, e.g, `loan_book`). You can chain more than one relationship type: of course, the target of a relationship type in the chain must be a *subtype* of the source of the following one. Note again that you may need to add a qualifier at the end of such a chain if the attribute your are interested in belongs to a supertype of the target of the last relationship type of the chain.

Let us build an artificial, but complete example using supertypes and identification functions (to see an interesting example with subtypes, see the Section called *Customising Labels*). Suppose that every person is actually owned by a `superhuman`, which is in turned owned by a `god`, which in turn is a subtype of `nonhuman`. Every `nonhuman` has a `name`. If we want to know the name of the god indirectly owning a person referred in a certain loan, we shall use

```
person_0.ownedby_superhuman->refersto_god->nonhuman.name
```

where we assumed that the identification function from `person` to `superhuman` is named `ownedby`, and that the identification function from `superhuman` to `god` is called `refersto`.

Following Arbitrary Relationship Types

ERW does not pose any limitation on the kind of relationship type that you can insert in an attribute specifier. This means, however, that using non-monodrome relationship types will lead to a situation where there are many values for the specified attribute, and that it may happen that a pointer along the path is `NULL`, in which case the resulting value will be the empty string.

For instance,

```
loan_person->lname
```

specifies the last names of all people who borrowed a certain book. Thus, the associated value is a *list* of strings, and not just a string. The meaning associated to such a specifier depends on the context (filtering, labelling, etc.) and will be fully explained later.

Custom Files

The first customisation layer is defined by means of customisation files. By default, they are contained in a directory named `custom` located where the run-time environment PHP scripts are. You can create a customisation file for each entity type, relationship type and instance of fileset type you would like to customise. The file

must be named as the identifier of the type postfixed with `.php`. A instance of fileset raising from an attribute *A* of an entity type *T* has identifier *T_A*, and then the same rule applies.

There is also a *default customisation file* that is used to change defaults throughout all types. It is named as the identifier of the root element of your ERL file postfixed with `.php` (in our library example, it would be named `library.php`). Each section on customisation specifies how to change the defaults using this file, whenever possible.

Your custom files are executed in the same environment of a hook: thus, you can tweak customisation using information such as the name of the user. Customisation files, however, should be light, as they are loaded each time a type is used.

All customisation in custom files is performed by setting certain keys of the array `$D`: the first key is always the name of the type that is being customised (e.g., `$D["person"]`). The custom files are loaded from higher to lower types, and this allows you to customise higher types *from the custom file of a lower type*. This is most useful: for instance, you could decide to customise an attribute of `person` in a different way when it appears in the form of a `subscriber`. To do so, you just need to put in the custom file for `subscriber` a suitable entry starting with `$D["person"]`.

Caution

Your custom files are executed as PHP scripts, but they must emit *no output*. In particular, always check that there are no blank lines after the processing-instruction delimiter `?>`.

The Type Array

The definition file of an entity type, relation type or fileset with attributes contains an associative array that specifies the type of each editable element, whether it is an attribute or a relation. A relationship type *R* from entity type *A* to entity type *B* is named *R_B* in the definition file for *A*, and *R_A* in the definition file for *B*.

The type array is named `$D[`*type*`]["type"]` (all information from definition files is stored in a associative tree contained in the `$D` variable). By assigning a certain ERW type to a variable, you can change the way it is edited. For instance,

```
$D["loan"]["type"]["duration"] = r;
```

will tell ERW that, in our library example, the duration should be chosen via a check-box rather than via a drop-down list.

Note: You can avoid quotes around `r`; it is predefined as a constant by ERW, so the lack of quotes will not generate any warning.

Customising Labels

Each time ERW has to display an entity, it uses a *label* for that entity. The label automatically set up is simply the content of the first attribute specified for the entity type, but it can be customised. The same holds for labels of relationships with attributes.

The array controlling the label of an element is `$D[`*type*`]["display"]`. More precisely, `$D[`*type*`]["display"]["fmtString"]` must be a `printf()`-like format string (it must be interpreted by PHP), whereas `$D[`*type*`]["display"]["field"]` must be an array of attribute specifiers and constant strings, whose values will be used together with the format string. Of course, the number of elements of

`$D[type]["display"]["field"]` must match the number of format specifiers in the format string.

Certain data, such as enumerative type or thousand-separated digits, are automatically formatted in the same way they would be presented to users in forms. In this cases, you should always use a `%s` directive in the format string. The `%d` directive can be used only with nonformatted numeric types.

> **Note:** ERW is usually able to distinguish whether a string is a constant or an SQL specifier. In case it gets confused, just start your constant string with `'`: the quote will not be printed, but ERW will know that you meant literally what follows.

Format String Extensions

The format string supports two useful extensions, which have been thought to handle the case of `NULL`values and multiple-value attribute specifiers.

If a directive has a `?` just after `%`, then the directive is *optional*. If the associated parameter is empty or `NULL` then the directive, the preceding one and the following one are marked as dead, and will not be actually printed (note that an optional directive cannot be marked as dead). This allows you to build complex specifications that add markup (such as brackets, hyphens and so on) depending on the presence of a certain parameter.

As an example,

```
$D["book"]["display"]["fmtString"] = "%s: %s %s%?s%s";
$D["book"]["display"]["field"] =
     array( "title", "author", " (", "isbn", ")" );
```

would display a book by showing the author followed by a colon, the title and then the ISBN code between parentheses, but *only* if the code has been specified.

> **Warning**
>
> There are situation (such as editing weak entities with ERW type `r1e`) in which ERW has no way to know the attributes of other entities. In this case, they are all set to `NULL`. Your format strings must always be written so that they handles these cases gracefully.

The other useful extension allows to format lists (of formatted values) using a given separator and additional formatting. More precisely, ERW accepts directives of this form:

`%`*`alignment`*`-width-`*`precision-length`*`[`*`separator`*`]`*`standard-format-specifier`*

When printing a directive of this form, ERW expects as corresponding field a multiple-value attribute. Each element of the list of values will be formatted using the *`standard-format-specifier`*. Then, the resulting strings will be concatenated using the given *`separator`* (which can also be delimited by `(`, `<` or `{`), and finally the resulting string will be printed using the given alignment, width, ecc.: essentially, the string will be formatted with a directive obtained by stripping the separator and whatever follows, and appending `s`.

As an example,

```
$D["book"]["display"]["fmtString"] = "%s: %s %[, ]s";
$D["book"]["display"]["field"] =
     array( "title", "author", "loan_person->lname" );
```

would display a book by appending to the title and author the list, separated by commas, of the people that borrowed the book (note however that in this case you could get doubles if the same person borrowed many times the same book).

Tabular Displays

Sometimes, you could prefer a rigidly aligned, tabular display in which each field occupies a fixed amount of space. This is easy to obtain using `printf()`'s sophisticated field width and alignment control. However, you will also need a fixed-width font, and a table header specifying the name of the columns. To obtain this effect, you can set the variable `$D[`*`type`*`]["display"]["head"]` to any string.

Virtual Attributes

When building labels, sometimes you can use *virtual attributes*, which do not really exist in the database, but are generated at run-time. These are the currently implemented virtual types:

MIMEtype

: Only for filesets; contains the MIME type of the file (it is derived from the filename extension).

original

: Only for filesets; contains the original filename if you are using original filenames.

fileSizeB
fileSizeKB
fileSizeMB
fileSizeKiB
fileSizeMiB

: Only for filesets; contains the size of the file in the given unit (note that K means 1000, Ki means 1024 and so on). It is a float, so you can format it using the standard `printf()` conventions.

Displaying Owners and Groups

ERW sets up two virtual relationship types, `owner_usr` and `share_grp`, that can be used as any other relationship type in an attribute specifier to display information related to an entity belonging to a type that has element-based authorisation.

Displaying Relationships

Display customisation allows you to specify a label using the attribute values of a relationship. The resulting label will be *juxtaposed on the right* to the label of the related entity when displaying a relationship in an entity form. For instance, in Figure 2 loans are displayed showing the label of the borrowed book (the related entity) followed by a label derived from the relationship attribute values (type of loan and dates). Thus, your labels for relationships must be written with this arrangement in mind (for instance, they would better start with a space).

However, ERW allows you also to edit *directly* relationships. In this case, you may want to specify a more complex label, possibly depending also on the attribute values of the entities related by the relationship.

To this purpose, ERW provides the `$D[type]["fullDisplay"]` customisation variable. A good example for `loan` is the following one, which would give the output shown in Figure 6:

```
$D["loan"]["fullDisplay"]["fmtString"] =
  "%-10.10s %-10.10s %-36.36s %10.10s %10.10s";
$D["loan"]["fullDisplay"]["head"] =
  "First name Last name  Book title                          Start date  End date";
$D["loan"]["fullDisplay"]["field"] =
      array( "person_0.fname", "person_0.lname",
             "book_1.title",
             "startdate", "enddate"
      );
```

Accessing subtypes

The *qualifier* part of an attribute specifier may be also used to access attributes of a subtype. Of course, the attribute could not exist at all, because the entity specified is not of that type: in this case, the value will be `NULL`.

Interestingly, we can use this fact to build labels that are conditional on the type of an entity. For instance, if we have an entity `document` with attribute `title` and subtypes `publication` (with mandatory attribute `journal`) and `techrep` (with mandatory attribute `number`), we can specify a label like:

```
$D["document"]["display"]["fmtString"] = "%s %s%?s%s%s%?d%s";
$D["document"]["display"]["field"] =
        array("title", "(", "publication.journal", ")",
                       "(", "techrep.number", ")" );
```

As a result, each document has an optional item between parentheses if it is a publication or a technical report: the journal or the number, respectively.

We can push further this technique exploiting the ubiquitous `id` attribute. Since subtyping is realised by set inclusion, an entity with a certain `id` is really of a more specific type only if the same `id` exists in a subtype. This can be exploited as follows:

```
$D["document"]["display"]["fmtString"] = "%s %s%?.0s%s%s%?.0s%s";
$D["document"]["display"]["field"] =
        array("title", "(", "publication.id", "Publication)",
                      "(", "techrep.id", "Technical Report)" );
```

The `id` field is used to trigger the `%?.0s` specifier, but because of the limitation on the output (`.0`), the `id` is not really printed. Rather, the strings `(Publication)` and `(Technical Report)` are only printed if the `document` is a `publication` or a `techrep`, respectively.

Local Customisation

It can happen that the standard way to label an entity does not work in all situations. The typical example is given by weak entities: suppose again you have editions for books, and that editions are weak entities whose owner is the respective book. Certainly, to label in general an edition you will use also fields from the owner, for example title and author, by means of attribute specifiers.

However, you will more likely access editions in the context of a specific book, using a suitable ERW type. In this case, however, you need just to show information about the edition, and not information about the owner, which is already currently displayed.

For this kind of situation, ERW offers *local customisation*. For each relationship type, you can specify a label for the related entities, which will override the standard one. The label is specified using the usual keys `fmtString` and `field`, but preceded by the key chain `$D[`*`type`*`]["display"]["`*`relationship-type_other-entity-type`*`"]`. For instance,

```
$D["book"]["display"]["regards_edition"]["fmtString"] = "%s, %d";
$D["book"]["display"]["regards_edition"]["field"] =
  array( "publisher", "year" );
```

will list editions (in the form of a book) just by their publisher and year (we are assuming that the identification function going from `edition` to `book` is named `regards`, and that `edition` has attributes named `publisher` and `year`).

Ordering Elements

By default, entities and relationships are ordered with respect to their identifier; this, however, is not very meaningful. You can specify the display order of entities and relationships by setting the variable `$D[`*`type`*`]["orderBy"]` to a string containing an SQL `order by`-like order specification: a list of comma-separated attribute specifiers, possibly followed by `asc` or `desc` (for *ascending* or *descending* order). For instance,

```
$D["book"]["orderBy"] = "book.author,book.title";
```

would order books by author and title, in ascending order. Instead,

```
$D["loan"]["orderBy"] = "loan.startdate desc";
```

would put in front the most recent loans, and

```
$D["subscriber"]["orderBy"] = "person.lname";
```

would order subscribers by last name.

Note that you cannot order absorbed relationship types, and that you cannot use non-monodrome relationship types in an order specification (in any case, it would not be meaningful).

Customising Filters

Each time ERW has to display a list of entities, relationships or files, it can provide a *filter* to allow the user to restrict the available choices. The default filter works on the content of the first attribute, but it can be customised. If several filters are specified, they work in boolean conjunction. Moreover, multi-value attribute specifier generate a filter that is satisfied if *any* of the values in the associated list satisfy the value in the filter.

To customise filtering, you specify a list of attributes over which filtering must be performed. Each attribute has a *level*, that is, a real number between 0 and 1 (inclusive, with default 1/2) that expresses how significant that attribute is with respect to filtering.

Each list displayed by ERW has a default level, too: it is 0 for main lists, 1/2 for selection lists and 1 elsewhere. For each list, ERW will display *only those filter whose level is greater than or equal to the list level*, in definition order. Thus, by default main lists and selection lists get all filters, whereas no filters are used elsewhere. Note that relationship types without filters inherit the filters of their source/target entity types.

The first array controlling filters is `$D[`*`type`*`]["filter"]`. More precisely, for each attribute specifier (the first key), you can specify an ERW type (second key "type"),

an optional label (second key "label"; if the label is omitted, the label of the attribute will be used instead) and an optional filter level (second key "level"; default 1/2). Presently the ERW type must be the default ERW type associated to the ERL type of the attribute, except for the ERL type a, for which the type t must be used. Since there is a default filter, and many levels of keys, it is usually a good idea to first reset the array and then fill one-by-one the desired entries.

As an example,

```
$D["book"]["filter"] = array();
$D["book"]["filter"]["author"]["type"] = t;
$D["book"]["filter"]["author"]["label"] = "Author(s)";
$D["book"]["filter"]["title"]["type"] = t;
$D["book"]["filter"]["isbn"]["type"] = t;
$D["book"]["filter"]["isbn"]["level"] = 0.1;
```

will allow to select a book by author, title *and* ISBN code in the main list, but just by author and title elsewhere. Moreover, the author filter will be labelled “Author(s)”.

You have the full power of attribute specifiers at your disposal. For instance, you can filter subscribers on their first and last name:

```
$D["subscriber"]["filter"] = array();
$D["subscriber"]["filter"]["person.fname"]["type"] = t;
$D["subscriber"]["filter"]["person.lname"]["type"] = t;
```

Multi-value attributes may be very useful, too, assuming you want filter out books that have been lent to someone in particular:

```
$D["book"]["filter"] = array();
$D["book"]["filter"]["loan_person->lname"]["type"] = t;
```

And, of course, you can access the source/target entity types of a relationship type, and their supertypes. For instance,

```
$D["loan"]["filter"] = array();
$D["loan"]["filter"]["person_0.fname"]["type"] = t;
$D["loan"]["filter"]["person_0.lname"]["type"] = t;
$D["loan"]["filter"]["book_1.author"]["type"] = t;
$D["loan"]["filter"]["book_1.author"]["label"] = "Author(s)";
$D["loan"]["filter"]["book_1.title"]["type"] = t;
$D["loan"]["filter"]["startdate"]["type"] = d;
$D["loan"]["filter"]["enddate"]["type"] = d;
```

would have the following effect: when editing loans, you would get a filter allowing to select loans by borrower’s first/last name, book author/title, and loan start/end date. On the other hand, the filters for selecting the loans of a certain person (in its form) would be the book title and the loan start/end date (analogously for a book).

Filtering on Owners and Groups

Must as in the case of labels, ERW sets up two virtual relationship types, `owner_usr` and `share_grp`. They behave as having type `r1`, and you can use them not only in specifiers, but also directly as attributes, using the type `r1s`. As a result, the filter will contain a drop-down list with available users (or groups).

Filtering Levels

The second array controlling filters is `$D[`*`type`*`]["filterLevel"]`. Its first key is a list type (as explained in the Section called *The Button Hook*), and its second key (which is to be specified only if the first key is not `main`) is of the form *`relationship-type_other-entity-type`* for relationships types, and *`attribute-name`* for filesets. It specifies a filter level (i.e., a real number between 0 an 1 inclusive) that overrides the default for any of the lists appearing in a form, allowing fine-grained customisation.

As an example,

```
$D["book"]["filterLevel"]["rel"]["loan_person"] = 0;
```

will set up all filters for searching persons who borrowed a book.

The default filter level can be changed in the default customisation file, using the key chain `$D["*"]["filterLevel"]` followed by a key giving the type of list (as explained in the Section called *The Button Hook*). Thus, for instance, `$D["*"]["filterLevel"]["rel"] = 0` would set all filters for all relationship lists.

Default Values

Sometimes, in particular with very large databases, you may want to present since the start a list which selects a subrange of items. to this purpose, ERW allows you to set default values for filters. Similarly to default value for attributes, you have to specify an SQL expression that will be evaluated to produce the initial filter value. The expression must be stored using the key `default` (which is as the same level as `type`, `label` etc.). For filters with operators, you can also specify the preferred operator using the key `operator`. For instance,

```
$D["loan"]["filter"] = array();
$D["loan"]["filter"]["startdate"]["type"] = d;
$D["loan"]["filter"]["startdate"]["default"] = "'2001-01-01";
$D["loan"]["filter"]["startdate"]["operator"] = ">";
```

would show at start loans that began in the current millennium.

Hooks

Hooks allows you to customise part of ERW's user-interface and database processing, or even to insert your own code in the middle of ERW's. They can be used to add your own custom buttons at the bottom of a list or to check that complex constraints on the content of the database fields are satisfied. Hooks are set up by setting suitable variables in the customisation file.

Writing hooks require some understanding of ERW's internals; you may also need to have a look at Chapter 10. A hook runs in a controlled environment, with a separate variable namespace from the rest of ERW. For your convenience, all relevant global ERW configuration variables are accessible; moreover, for each variable a local copy *without* the `_ERW_` prefix is defined.

Some hooks may pass you arguments: they will be contained in the associative array `$arg` (the keys and the meaning of the values will vary from hook to hook).

> **Caution**
>
> The local copies are really copies: if you modify them, the corresponding global ERW variable will *not* be modified. In some cases, this is not what you want.

Besides the global configuration variables, the following ones are available:

$_ERW_db

: A PEAR database connection. You can use all PEAR standard methods on it, or you can directly use ERW's internal versions. Note that usually accessing types not owned by the current entity could give rise to a database error.

$_ERW_userId

: The `id` of type `usr` corresponding to the current user, or 0 if no user has been authenticated, or if the authenticated user is not in `usr`.

$_ERW_auth

: An associative array with keys `select`, `update`, `insert` and `delete` that describe the current permissions.

$_ERW_locale

: The locale that has been negotiated with the browser.

$_ERW_lang

: The language part (the first two letters) of the locale that has been negotiated with the browser.

Note again that you can access local copies of these variables under the names `$db`, `$userId`, and so on.

The Lock Hook

The lock hook is used during database update and deletion phases. It is set by assigning to the variable `$D[`*type*`]["lock"]` an array containing the names of tables that must be locked together with the neighbourhood of the current entity. It allows you to set up things so that you can atomically check and possibly update your own support tables during an atomic ERW transaction.

The Pre-update Hook

The pre-update hook is called *before* actually updating the content of a row (or inserting a new row). It is set by assigning to the variable `$D[`*type*`]["preUpdate"]` the name of a PHP file. Note that the file is relative to the server directory where PHP scripts are installed. If it exists, the file is included and its return value (that you can set using the `return` instruction) is checked. If it is the empty string (or `null`), the check is considered successful. Otherwise, the returned string is considered to be an error message: the changes will not be applied, the string will be displayed and the user will be presented again with the form he or she was trying to submit. Note that in the case there are supertypes, pre-updates hooks are called starting from the maximal supertypes (i.e., supertypes that have no proper supertype) and going down to the actual type. Pre-update hooks for embedded (entity, relationship and fileset) types are executed afterwards.

To help in writing parametric scripts, the run-time envinronment sets up four global variables:

`$mainType`

The (entity or relationship) type of the element modified or created by the form currently processed.

`$mainId`

The numerical identifier (primary key) of the element that caused the current hook invocation, or 0 in case the element is new.

`$type`

The (entity, relationship or fileset) type of the element modified or created by the hook currently processed. This might differ from `$mainType` because of embedded editing of relationships, weak entities or filesets.

`$id`

The numerical identifier (primary key) of the element that caused the current hook invocation.

During the execution of the pre-update hook, you can access a read-only global variable `$q` (by declaring it as such or using the `$GLOBALS` array) that contains the query that it is going to be written. It is an associative array with two indices: the first index is a entity type identifier, and the second one an attribute identifier (note that in the case of subtypes, more than one entity type could be involved). The corresponding value is the *quoted value* that it is going to be written. By this we mean that strings are actually quoted and escaped, and should be passed through the `ERW::unquote()` function before being examined. Note that in almost all non-mandatory fields one of the possible values is `NULL`.

During the execution of the pre-update hook, all types in relation with or owned by the current entity type are locked. Queries outside this set could give rise to an error. However, you can use the lock hook to lock additional tables.

Caution

You should not try to change the locks in the pre-update hook. To maintain integrity, it is necessary that the pre-update and the update phase are all executed under the same lock.

Example 4-1. Checking that a title is long enough

In our library, no book can have a title with less than two characters. Thus, we set in the custom file of the `book` entity set the line

```
$D["book"]["preUpdate"] = "bookPreUpdate.php";
```

and fill the file `bookPreUpdate.php` with

```
<?php
global $q;
if (strlen(ERW::unquote($q["book"]["title"])) < 2) return "Title too short!";
else return "";
?>
```

The Post-update Hook

The post-update hook is called *after* actually updating the content of a row (or inserting a new row). It is set by assigning to the variable `$D[`*type*`]["postUpdate"]` the name of a PHP file. Note that the file is relative to the server directory where PHP scripts are installed. If it exists, the file is included (you have at your disposal

the same variables as in the pre-update hook). Usually at this point some additional update or check of the database is performed.

Post-updates hooks are run initially for embedded (entity, relationship and fileset) types. Then, the hooks for the element modified or created by the form are executed; in the case of supertypes, hooks are invoked from the current type upwards (i.e., exactly in the dual way w.r.t. pre-update hooks). This allows you to impose complex contraints to relationships (even belonging to different types), as you can check them looking at the current database content during the post-update hook.

After your custom checks and changes, you can return `null` to specify that you are satisfied with the current situation, or a string; in the latter case, the transaction will be aborted and the string will be displayed to the user as an error message (much like a pre-update hook).

Warning

Transaction support is not currently available using MySQL. Returning a non-`null` value will lead to unpredictable results.

During the execution of the post-update hook, the database is still locked. This can be changed, however, if necessary. The programmer must take into consideration that changing a lock implies that another transaction could sneak in (maybe deleting the row just inserted!). If the lock is not modified, the post-update hook is executed atomically together with the pre-update hook and ERW's update phase.

The Deletion Hooks

Hooks exists also to track entity deletions. They are activated, similarly to the pre-update hook and to the post-update hook, by assigning the variables `$D[`*`type`*`]["preDelete"]` and `$D[`*`type`*`]["postDelete"]`. The behaviour is in all ways analogous to the update hooks, except that the only information available is the global variable `$id` containing the primary key of the entity that it is going to be deleted.

The Post-Authorisation Hook

The post-authorisation hook is called just after the authorisations for the current element or type have been computed. It is set by assigning to the variable `$D[`*`type`*`]["postAuth"]` the name of a PHP file. Note that the file is relative to the server directory where PHP scripts are installed.

When the hook is run, the associative array `$arg` will be loaded with three keys: `type`, providing the type for which the authorisation is being computed, `id`, providing the element of the given type for which the authorisation is being computed (it will be 0 if the authorisation is not element-specific), and `auth`, providing the current authorisation array computed by ERW. The latter associates a boolean values to the keys `select`, `update`, `insert` and `delete`; each value specifies whether the corresponding privilege is granted for the current element or type. Moreover, it associates a string to the keys `selectDenied`, `updateDenied`, `insertDenied` and `deleteDenied`: the string is the error message produced when the corresponding privilege is denied. Note that in the case of a denied update privilege in the presence of select privileges, the message will be displayed but the user will be anyway presented with a read-only form. Moreover, an empty string will cause no error message to be displayed. You can modify the defaults for these messages by modifying a suitable configuration variable.

During the execution of the post-authorisation hook, you can just assume that the given type and all authorisation-related tables are locked. Queries outside this set could give rise to an error.

Example 4-2. Old book entries cannot be modified

In our library, books older than 1950 have read-only entries: their data cannot be modified, and they cannot be lent. To obtain this effect, we can just check the publication year in a post-authorisation hook and clear the update privileges in a file named `bookPostAuth.php`:

```
<?php
if (ERW::getOne("year from book where id=".$arg["id"]) < 1950) {
   $arg["auth"]["update"] = false;
   $arg["auth"]["updateDenied"] = "You can just browse this entry!";
}
?>
```

Of course, the custom file for `book` will contain a line

```
$D["book"]["postAuth"] = "bookPostAuth.php";
```

Note that you can also access the usual variables available in a hook: if the restriction does not apply to the users of group 1, then you can modify the hook as follows:

```
<?php
if ($userGrp != 1 && ERW::getOne("year from book where id=".$arg["id"]) < 1950)
   $arg["auth"]["update"] = false;
?>
```

The Selection-load Hook

The selection-load hook allows to restrict the elements that will be displayed to the user in selection lists. You can specify for each relation adjacent to the current entity a set of additional tables to insert into the `from` part of the SQL clause that will be used to fill the list, and also a set of constraints that will be joined to the `where` part. For instance,

```
$D["person"]["loadSel"]["loan_book"]["from"] = array("loan");
$D["person"]["loadSel"]["loan_book"]["where"] =
    array("loan.id1_book=book.id", "loan.enddate is not null");
```

would restrict the books that can be borrowed to those that are not currently on loan.

> **Caution**
>
> This hook is deprecated, and may not be supported in future versions of ERW

The Button Hook

ERW provides editing for entities and relationships, but there are other services that you might like to provide. For instance, a printed version of an invoice, or a pretty-printed version of the list of entities. The button hook allows you to add buttons at the bottom of lists that invoke JavaScript actions, which could in turn open new windows that would be filled by PHP code.

The button hook is activated by defining in the custom file of an entity (or of a fileset) an associative array named `$D[`*`type`*`]["button"]`. The first index is the kind of list you want to attach the button to. Possible value are `main`, meaning that you will add

buttons to the main list; `sel`, meaning that you will add buttons in all lists used to select entities to relate to the current entity; `rel`, denoting a list of relationships; and `fset`, denoting the list of files in a fileset. The value of the array for each index must be again an array where each element describes a button. Each element is in turn an associative array with keys `label`, `name`, `help` and `events`. The values of `label` and `name` will be the name and the label of the button, whereas the value associated to the `events` key is an associative array, having as keys event attributes (`onclick`, `onmousedown` etc.) and as values fragments of JavaScript code to be executed. The value of the `help` element will be displayed as a tooltip when the mouse passes over the button.

Each relevant HTML element in a page generated by ERW has a well-defined identifier. The main list for the entity *A* is named `m_A`. There are a number of useful functions that let you access the JavaScript state.

Once you have the data you need, the best thing to do is usually to open a new window and fill it with a suitable PHP script. To this purpose, the function `openServiceWindow()` is available: it opens a windows without navigation buttons and loads it with the URL given as first parameter; the second parameter is a symbolic name for the window that is purely informative.

Example 4-3. Pretty-printing a book data.

Our librarian now asks for the possibility of pretty-printing the data of a book, and of printing a report of all lends for a certain book. It is easy to prepare a couple of PHP scripts doing that and outputting HTML. By adding the following to the customisation file for the entity `book` these features will also be easily accessible from the main list:

```
$D["book"]["button"]["list"] = array(
  array(
    "label" =>
      "Print data",
    "name" =>
      "data",
    "help" =>
      "Print all data related to the selected book",
    "events" =>
      array(
        "onclick" =>
          "if ((i = getSelectedId('m_book')) < 0)
             alert('No book selected');
           else openServiceWindow('data.php?id='+i, 'data')"
    )
  ),
  array(
      "label" =>
        "Print history",
      "name" =>
        "history",
    "help" =>
      "Print all loans related to the selected book",
    "events" =>
        array(
          "onclick" =>
            "if ((i = getSelectedId('m_book')) < 0)
               alert('No book selected');
             else openServiceWindow('history.php?id='+i, 'history')"
    )
  )
);
```

Of course, the scripts `data.php` and `history.php` must have a look at the variable `$id` and produce the right data.

The Button Inhibition Hook

Sometimes you may want to keep the user from doing certain things, such as creating new elements, maybe providing your own buttons to do some custom processing (using the button hook). ERW provides a button inhibition hook that allows you to select which of the standard buttons will be displayed. The inhibition hook is activated by defining in the custom file of an entity an associative array named `$D[type]["inhibit"]`.

If there is an entry of `$D[type]["inhibit"]` whose first key is `ok`, `save`, `clone` or `cancel`, the corresponding form button will not be displayed.

Moreover, if there is an entry of `$D[type]["inhibit"]` whose first key defines the type of list, as described in the Section called *The Button Hook*, and whose second key is `new`, `modify`, `delete`, `select`, `cancel`, `add`, `remove`, `apply`, `reset`, or `chown`, then the corresponding button will not be displayed.

Inhibition can be set in the default customisation file. The key chain is as above, but you should use `"*"` for the table name. Thus, for instance, `$D["*"]["inhibit"]["clone"]` eliminates the cloning button from all forms, whereas `$D["*"]["inhibit"]["sel"]["new"]` eliminates the new element button from all selection lists.

Example 4-4. User front-end

Our librarian wants the users to be able to browse the database using ERW. Of course, users should be able to scan the available books, and maybe some custom button should make them able to print data about them, but we do not want to confuse them with other buttons. This can be obtained by adding the following to the customisation file for the entity `book`:

```
$D["book"]["inhibit"]["list"]["new"] = true;
$D["book"]["inhibit"]["list"]["modify"] = true;
$D["book"]["inhibit"]["list"]["delete"] = true;
```

Note that using `true` as a value is purely incidental—any other (logically true) value would have done the job.

The Form-Loading Hook

When creating a form for editing, ERW must decide whether it is the case to provide the user with a custom form. The chosen custom form depends on a *key* (by default, the name of the type of the element), which however can be set depending, say, on the user, using the form-loading hook.

The form-loading hook is activated by assigning to the variable `$D[type]["loadForm"]` the name of a PHP file: the string returned by the file will be used as a key to load the form (see the Section called *Form Resolution*).

Custom Forms

Custom files are very useful, but they cannot, for instance, change the order in which elements are displayed. *Custom forms*, instead, are a much powerful feature that allows to structure freely ERW's editing forms.

A custom form is defined by an XML document in ERF. Once the form has been created, it must be compiled using `ERtool` and the resulting PHP file must be installed in the custom form directory (which is defined by a suitable configuration variable).

Form Resolution

Forms may present the user with information in different order, different language, different level of authorisation, and so on. To provide maximum flexibility, ERW defines a *form resolution* procedure that determines the form that will be actually loaded in a certain circumstance.

First of all, form resolution is based on a *key*, which by default is given by the name of the type of the element the form will be used with. Thus, the default key for the form of a book is `book`. Once the key is determined, ERW tries first to find a form named *`key`*`.current-locale.php`, e.g., `book.it-IT.php`. If the form cannot be found, ERW will try *`key`*`.current-language.php`, e.g., `book.it.php`. Finally, ERW will try to load *`key`*`.php`, e.g., `book.php` (this process is very similar to the one used by `Apache` to load language-dependent pages).

There are two ways to change the default key: the form key hook, which lets you choose the key using PHP code, and the form ERF attribute, which sets the form key for an embedded form.

ERF: an Entity-Relationship Form Language

ERF is an XML-based language that is used to describe custom forms.

Note that there are two kind of forms: *embedded* and *non-embedded*. Non-embedded forms are the standard ones: using them, you alter each element of an ERW database. Embedded forms, on the other hand, allow you to edit relationships directly into the form of an entity, or to edit weak entities directly into the form of the owner. The main difference is that non-absorbed relationship types and filesets cannot be part of an embedded form.

A Quick Tour

Before giving the reference to ERF elements, it is useful to have an informal discussion of what you can describe in ERF.

ERF documents looks really like XHTML documents. Indeed, ERF contains a selection of XHTML elements (e.g., `div`, `style`, ecc.) and a custom form looks like a simplified XHTML page. The interesting point, however, is that placing suitably `input` and `list` elements you can place freely ERW input controls and lists along the page. If you want to have a look at sample ERF custom forms, `ERtool` can produce for each type a form giving the standard look. For instance, the form produced for a person looks as follows:

```
<?xml version="1.0"?>
<!DOCTYPE erf PUBLIC "-//DSI//DTD ERF V1.0//EN" "file://erf.dtd">
<erf ref="person">
  <input ref="fname"/>
  <input ref="lname"/>
  <list ref="loan_book"/>
</erf>
```

A corrsponding customised form has been shown in the Section called *A Simple Example* in *Preface* (more precisely, in Figure 7).

Elements

In this section we list all ERF elements, giving in detail their usage and meaning. First of all, there are XHTML-like elements `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `div`, `fieldset`, `p`, `span`, `em`, `strong`, `i`, `b`, `sup`, `sub`, `samp`, `code`, `var`, `legend`, and `img`, which possess `id`, `class` and `style` attributes, and an element `style` with attribute `type`. The element `img` has also attributes `src`, `alt`, `longdesc`, `width`, and `height`. Finally, there is an element `a` with an additional attribute `href`, but note that for obvious reasons the linked page will be always opened in a new window. For the meaning of these elements and attributes, please refer to the HTML or XHTML documentation.

erf

Name

`erf` — A whole custom form

Synopsis

The `erf` element is the top-level element of an ERF document. It contains any other ERF element.

Attributes

ref

: The type this form refers to.

emb

: A boolean denoting whether this form is embedded or not. If not specified, it is assumed to be false.

input

Name

`input` — An input control

Synopsis

The `input` element defines where the input control(s) for a certain attribute will appear in the form.

Attributes

ref

: The ERL attribute this element refers to.

label

An alternative label for this attribute. If not specified, the localised version of the label specified in the ERL file will percolate through.

help

Alternative help text for this attribute. If not specified, the localised version of the help text specified in the ERL file will percolate through.

type

An ERW type. If not specified, the type specified in custom files will percolate through.

size

For the types for which it is meaningful (e.g., text), the size of the HTML input control.

readonly

If true, this attribute cannot be changed by the user.

tabindex

This attribute is in all ways identical to the corresponding HTML attribute, and will be passed to the HTML input control.

list

Name

`list` — An input list (or set of lists)

Synopsis

The `list` element defines where the list(s) for a certain relationship or fileset type will appear in the form (this includes also transpose of identification functions, in which case you are really editing owned weak entities).

Attributes

ref

The fileset or relationship type this element refers to.

label

An alternative label for this (set of) lists(s). If not specified, the localised version of the label specified in the ERL file will percolate through.

help

Alternative help text for this (set of) list(s). If not specified, the localised version of the help text specified in the ERL file will percolate through.

type

An ERW type. If not specified, the type specified in custom files will percolate through.

form

A form key that will be used in the form-resolution process to load the embedded form for this (set of) lists(s).

readonly

If true, this fileset or relationship type cannot be changed by the user.

tabindex

This attribute is in all ways identical to the corresponding HTML attribute, and will be passed to the HTML input control.

Chapter 5. Authentication and Authorisation

ERW is designed to work well in complex intranets in which several users access data with different privileges. This requires two basic services, that is, authentication and authorisation. By default, ERW delegates completely the authentication process to the HTTP server. The system administrator should set up directory access, accounts and passwords so that PHP can fill its internal variables[1].

Custom Authentication

If you prefer to set up your own authentication system, you have just to set the `$_ERW_authenticate` variable in the main configuration file. This variable must be set to the name of a PHP file containing code that will perform authentication, usually looking at the content of the `$_SERVER["PHP_AUTH_USER"]` and `$_SERVER["PHP_AUTH_PW"]`. The code will run in the same environment of a hook, must end with a `return` statement, and must return `true` or `false`, depending whether the user is authenticated or not. A nonauthenticated user will be presented with the typical HTTP basic authorisation dialog; you can set the basic realm (that will be shown to the user) using the configuration variable `$_ERW_basicRealm`.

> **Note:** You can extend at will the default authorisation-related entities contained in `auth.xml`, for instance adding a `password` attribute (with ERW type `pw`) to the entity `usr`.

For instance, assuming that you added a `password` column to the `usr` table, you can use the following code:

```
<?php
return $db->getOne("select COUNT(*) from usr where ".
                   "login=".ERW::quote($_SERVER["PHP_AUTH_USER"]).
                   " and password=".ERW::quote($_SERVER["PHP_AUTH_PW"]));
?>
```

Put this code in a file named, say, `authenticate.php` located in the main server directory, set `$_ERW_authenticate` to `"authenticate.php"` in the main configuration file and you're done.

The Authorisation System

The permission system used in ERW follows strictly the UN*X conventions, and if you are used to those conventions you will find it very natural. To enable it, you must include the XML entity `&ERWauth;` in your ERL file. This will include the XML file `auth.xml`, which generates a few tables that ERW uses to keep track of authorisation information. The table definitions can be extended at will (for instance, providing a password field), as long as the relational structure and the provided attributes are left intact. If you do not include `&ERWauth;`, everyone can do everything.

> **Warning**
>
> Even if you do not want to use authorisation, you cannot use the name `type` for a type, as it is used to detected the presence of the authorisation system.

First of all, there are entities types `usr` and `grp` (with obvious meaning), related by `belongs` and by `primary`. In this way you can set up accounts (for each account you must specify a mandatory real name and an optional e-mail address) and groups

(to organise authorisation information). In particular, for each user you can specify a *primary* group, which plays a special rÃ´le.

Then, the entity type `type` is used to set up authorisations. Each entity in `type` has a name, which should be the name of an entity, relationship or dynamic enumerative type. If a type is not listed in `type`, then everyone can do with it whatever is granted by the global privileges (which by default are all, but this can be changed in the configuration file). As soon as a type is entered into `type`, however, it becomes accessible onl id="pecified users"y to the specified users and groups (it is easy to forget to register a type in `type`: however, there is a script that can do the check for you).

For each type you can separately specify (globally or on a user/group basis) *select*, *update*, *insert* and *delete* privileges. Users holding select privileges can list and inspect elements of a given type; users holding update privileges can modify elements of that type; users with insert privileges can add new elements of that type; users with delete privileges can delete elements of that type.

As explained in the definition of the `ent` element, you can specify that you want each entity of certain type to have an owner (and a group), that is, that you want element-based authorisation. In this case, the owner of an entity has some additional privileges on the entity (even if there are no global privileges). More precisely, when a user holding insert privileges on a entity type creates a new entity, the entity will have the user as owner and the user's primary group as group. These two assignments determine the additional privileges.

Each type has separate user select, group select, user update, group update, user delete and group delete privileges. The meaning is rather obvious: for instance, if user update is on, then users have update privileges on the entities they own, whereas if group delete is on, then users belonging to the group of an entity (or having has primary group the group of the entity) have delete privileges on the entity. Moreover, each set of authorisations granted to a user or a group may add typewise or elementwise additional privileges (e.g., a user authorisation may grant to a certain user for a certain type just user select privileges). In the end, a user gets all authorisations granted typewise, plus all elementwise authorisations granted to the user itselfs or to one of the groups the user belongs to.

To allow more flexibility, even if user and group select privileges are granted, you can tune in a finer way the entities shown to the user as explained in detail in the Section called *The Selection Level Tuning*.

> **Important:** Once you have update privileges on an entity, *you have the same privileges on all incident relationship*, that is, you can insert, delete or update any relationship involving the entity. This is natural, as relationships are essentially properties of the entity. Note, however, that if you edit a relationship type directly the permission you get are just those given for that type.

Some care must be taken when you deny global select privileges on a type: when you edit a relationship type from an entity form, you should have some select privileges for all entities adjacent by means of an editable relation (because ERW has to list elements for you to select them).

Changing Owner

If you have update privileges on a *whole entity type*, you can change the owner of its entities. If you have update privileges on an entity you can assign a group to the entity.

> **Note:** Unless you have update privileges on the type, you cannot give ownership to someone else, and you cannot choose a group the owner does not belong to.

The Selection Level Tuning

Each list displaying entities whose type has element-based authorisation may feature a distinct *selection level*. This level allows to set independently whether the user can see the entities it owns, the entities having its primary group, or the entities having a group the user belongs to.

The mechanism is the same used when doing customisations. There are three constants, `ERW_SELECTLEVEL_USER`, `ERW_SELECTLEVEL_PRIMARY`, and `ERW_SELECTLEVEL_GROUP`, that represent the three levels available. They are bit masks, so, for instance, you can write `ERW_SELECTLEVEL_USER | ERW_SELECTLEVEL_PRIMARY` to mean that you want to list entities the user owns or entities whose group is the user's primary group.

There are variables you can set in the default customisation file, using the key chain `$D["*"]["selectLevel"]` followed by a key giving the type of list (as explained in the Section called *The Button Hook* in Chapter 4). Thus, for instance, `$D["*"]["selectLevel"]["list"] = ERW_SELECTLEVEL_USER` sets the default for main entity lists to entity owned by the current user.

For a finer tuning, you can set separately the level for each list. The relevant variable is `$D["`*`table`*`"]["selectLevel"]`. Its first key is a list type, and its second key (which is to be specified only if the first key is not `main`) is of the form *`relationship-type_other-entity-type`* for relationships types.

As an example, assuming that we required element-based authorisation for `person`, and that we did not grant table-wise select privileges,

```
$D["book"]["selectLevel"]["sel"]["loan_person"] = ERW_SELECTLEVEL_USER;
```

will force only persons we own to be listed for creating loans.

Restrictions on Element-Based Authorisation

Element-based authorisation is useful, but it can easily lead to inconsistencies. For instance, if *`A`* is a *`B`* and you require element-based authorisation for both entity types, you could end up with a different owner for the same entity when you see it as of either of the two types. Fixing a choice (i.e., the most specific type) doesn't help much, as you would have redundant (and possibly nonvalid) information in *`A`*.

The solution chosen in the design of ERW is based on two principles:

1. If an entity type has element-based authorisation, all its subtype have it, too.
2. Only root types may have element-based authorisation.

A *root type* is an entity type that is maximal (i.e., it has no proper supertype) and such that all its subtypes have exactly one maximal supertype. In other words, a root type is the maximum type of the type hierarchy it belongs to.

For instance, if *`A`* is a *`B`* and also a *`C`*, then *`B`* and *`C`* are maximal, but they are not root types. So no type in this hierarchy may have element-based authorisation.

If, however, we have an additional type *`D`* such that both *`B`* and *`C`* are *`D`*, then *`D`* is a root type, and by requiring element-based authorisation for *`D`* we will get it for all subtypes of *`D`*.

In practise, if you do not have type hierarchies you can use element-based authorisation at will. If you have type hierarchies, you can use it only on hierarchies having a maximum type. If you really need several maximal types, you can always add a fake

root type without attributes having all maximal types as direct subtypes (and maybe make it abstract).

Notes

1. Of course, nothing prevents you from using the post-update hook to set up special forms that will actually make a user able to manipulate the server password file.

Chapter 6. Transaction Support

ERW provides partial support for SQL transactions and constraints. The support does not work presently for MySQL.

The usefulness of SQL-based constraints is evident if you access your database with different clients: in this case, including constraints at the SQL level avoids duplicating efforts.

ERW runs all update and delete operations inside a transaction. In case of an SQL error, the transaction is rolled back. The same happens when a post-update hook or a post-delete hook returns a non-`null` value.

However, to avoid cluttering the code logic, SQL errors cause a log to be written, and no further action being taken. This usually results in the user seeing an empty form, or a form with empty lists.

The notable exception, however, is the insertion or update of entities, and the commit phase at the end of an update or delete operation. In this case, errors lead to presenting again to the user the form that was submitted, displaying the error message coming from the database (unfortunately, the message coming from the database is often not exactly explicative).

Thus, you can easily add SQL constraints to your database, if they are checked during one of the phases above. For instance, deferred constraints are checked during the commit phase; thus, they are handled correctly by ERW.

The endorsed way of writing SQL-based constraints is thus to write SQL views and constraints that incorporates your checks, and call the functions from a post-update or post-delete hook.

Chapter 7. Installation and Configuration

Installing ERW is not a very complicated operation. However, there are several details that must be set up correctly in the environment for ERW to work.

The Run-Time Environment

The first step toward using ERW's powerful editing forms is installing all PHP scripts coming with the distribution (the content of the `php` directory) somewhere where they can be served by `Apache`. These scripts contain the run-time environment that produces the editing forms starting from the definition files produced by `ERtool`.

More precisely, the file `ERW.php` and the directory `ERW` can be installed in any place where PHP searches for source files (i.e., `/usr/share/php`), whereas the remaining files must be placed in a directory served by `Apache`. A reasonable option is to install all files in a generic place (e.g., `/usr/share/ERW/php`) and then create symbolic links, but in this case you must be sure that the Apache `FollowSymLinks` option is on.

The second step is producing definition files with `ERtool` and making them accessible to the run-time environment. As a first try, you should place the definition files in a directory named `defs` located in the same directory of the scripts (it is the place where definition files are searched for by default). As a first access method, you can change the name of `index.orig.php` file to `index.php`: it produces a list of links to the pages that allow you to choose and edit elements of the database. However, you should devise some custom access page that fits your needs.

The `ERW-conf.php` Initialisation File

All components of the run-time environment source initially the file `/etc/ERW-conf.php`, and a file named `ERW-conf.php` in the current directory. These files must contain definitions for a number of parameters; they are normal PHP files, so you can, for instance, use conditionals to provide different parameters on different virtual hosts. A sample file named `ERW-conf.orig.php` is provided for your convenience.

The following variables are defined in the file above: note that, following the PEAR convention, their name always start with `_ERW_`.

`$_ERW_erlId`

This string must match the identifier given to the `erl` element (see **erl**).

`$_ERW_dbType`

The type of the database you are using. Any PEAR specifier (`mysql`, `pgsql`,...) will do the job.

`$_ERW_dbOptions`

If this variable is defined, it will be passed as second argument to the PEAR `DB::connect()` call. Currently, the only allowed value is `true`, which requires persistent connections to the database, if available.

`$_ERW_dbHost`

The name of host running the database server.

`$_ERW_dbName`

The name of the database that should be used.

`$_ERW_dbUser`

The user name with which to access the database.

`$_ERW_dbPassword`

The password to access the database as the user `$_ERW_dbUser`.

`$_ERW_defsPath`

A relative or absolute path specifying the directory where the `ERtool`-generated configuration files are stored.

`$_ERW_customPath`

A relative or absolute path specifying the directory where the custom configuration files are stored.

`$_ERW_formsPath`

A relative or absolute path specifying the directory where the custom forms are stored.

`$_ERW_utf8`

If set to true, this variable enables UTF-8 support.

`$_ERW_htmlArea`

If set to a string, this variable enables HTMLArea support. The string must be a path (which can be relative to the document root) leading to the HTMLArea installation; usually, installing HTMLArea in the document root in a directory named `htmlarea` and setting this variable to `htmlarea/` is a good choice.

`$_ERW_fckEditor`

If set to a string, this variable enables FCKeditor support. The string must be a path (which can be relative to the document root) leading to the FCKeditor installation; usually, installing FCKeditor in the document root in a directory named `fckeditor` and setting this variable to `fckeditor/` is a good choice.

`$_ERW_jsCalendar`

If set to a string, this variable enables JSCalendar support. The string must be a path (which can be relative to the document root) leading to the JSCalendar installation; usually, installing JSCalendar in the document root in a directory named `jscalendar` and setting this variable to `jscalendar/` is a good choice.

`$_ERW_localePath`

An *absolute* path specifying the directory where gettext localisation files are stored. If you need to specify a directory relative to the document root of the HTTP server, use the PHP variable `$GLOBALS["DOCUMENT_ROOT"]`.

`$_ERW_fixedLocale`

A fixed locale, which will override the one provided by the browser. Set it to the empty string if you want to use the locale provided by the browser.

`$_ERW_defaultLocale`

A default locale, to be used instead of `en-US` when the browser does not provide a locale, and no fixed locale has been set.

`$_ERW_decimalPoint`

The character for the decimal point (i.e., the character that separates the integer and the fractional part of a number).

`$_ERW_decimalComma`

The character for the decimal comma (i.e., the character that separates triples of digits in the integer part of a number).

`$_ERW_dateFormat`

A number denoting the date format you prefer. It can be `ERW_DATE_ISO`, `ERW_DATE_EU`, or `ERW_DATE_US`. See also Chapter 11.

> **Warning**
>
> Do *not* make on-the-fly changes to this value—changing it should be considered as severe as installing a new version of ERW. Browser caching of JavaScript code can cause pernicious inconsistencies (see also the Section called *Caching* in Chapter 3.

`$_ERW_supportedLocales`

A PHP array of locales for which translations are available. HTTP language negotiation will be used to select the most appropriate one. This variable defaults to `array("en-US")`.

ERW will try to match the locale requested by the browser with one of the supported ones. As a last resort, it will use a country-specific language (such as `en-US`) for a generic language (such as `en`). Thus, the order in the array is important: if you specify `array("en-UK", "en-US")`, upon a generic request for `en` the user will get the `en-UK` locale.

`$_ERW_fileBasePath`

An absolute path indicating the directory where all files relative to the database are stored. A hierarchy of subdirectories (one for each entity, and then one for each fileset) will be created automatically when files are uploaded.

Important: This directory (and its subdirectories) must be writable by the user under which `Apache` runs (usually `apache` or `nobody`).

`$_ERW_authenticate`

The name of a PHP file, containing code for custom authentication. See Chapter 5.

`$_ERW_globalPrivileges`

A conjunction of the bit masks `ERW_SELECT`, `ERW_UPDATE`, `ERW_INSERT` and `ERW_DELETE` that gives the global privileges. The default value is `ERW_SELECT | ERW_UPDATE | ERW_INSERT | ERW_DELETE` (i.e., all privileges to everybody). See the Section called *The Authorisation System* in Chapter 5.

`$_ERW_authDenied`

Default error messages for denied authorisations. This array must associate to the keys `selectDenied`, `updateDenied`, `insertDenied` and `deleteDenied` corresponding error messages (an empty message will silence the error display). The messages can be further customised using the post-authorisation hook.

`$_ERW_basicRealm`

If you set up custom authentication, this is the basic realm that will be shown to the user in the authentication dialog.

`$_ERW_logLevel`

A number denoting the logging level required. It can be `ERW_DEBUG`, `ERW_WARNING`, `ERW_INFO` or `ERW_DEBUG`.

The following variables set up defaults for the user preferences. They can be recorded in a cookie, so each user can have a different setup.

`$_ERW_fontSize`

The font size, in points. 0 corresponds to "medium" in CSS language (i.e., the user preferred size).

`$_ERW_numRows`

The number of rows for fixed with elements. This variable is an associative array, indexed by `a` (for `TEXTAREA` elements), `l` (for main lists), `s` (for selection lists in relation editing), `r` (for relationship lists), and `f` (for filesets).

`$_ERW_uploadFrameHeight`

The number of pixels of the inline frames containing the HTML elements managing file uploads for `TEXTAREA` elements and filesets. If it is possible to determine at runtime the correct width and height of these frames, ERW will do it, but in case something goes wrong, the starting height can be tuned by the administrator, or even by the user.

`$_ERW_useFieldsets`

ERW uses the `FIELDSET` element to group items. In some very old browsers such elements may cause layout problems. In this case, you can set this variable to false.

`$_ERW_useOriginal`

Since 1.0.10, ERW keeps track of the original filename of each file in a fileset. This feature, however, is disabled if not enabled explicitly by setting this variable to true, as it is not compatible with previous versions. The default configuration file sets this variable to true, so that original filenames are enabled by default in new installations.

Configuring PHP

There are also a few option that must be set up in PHP for ERW to work:

No magic quotes.

All "magic quote" options *must* be turned off. More explicitly,

```
magic_quotes_gpc = Off
magic_quotes_runtime = Off
magic_quotes_sybase = Off
```

To be true, only the first line is strictly necessary: ERW can turn off the other settings internally.

The `;` separator.

ERW uses the W3C-endorsed separator `;` for separating the query part of a URL. The original symbol, the ampersand, was not well thought, as it should be really written `&`. You should add the line

```
arg_separator = "&;"
```

or

```
arg_separator.input = "&;"
```

(depending on your PHP version) somewhere in you PHP configuration file.

Include paths

PHP must be able to retrieve both PEAR scripts and ERW run-time scripts. A reasonable setting (which however may very depending on your operating system and distribution) is

```
include_path = ".:/usr/share/pear:/usr/share/php"
```

if you installed the scripts as suggested in the `README`.

Logging

ERW is a complex system, and heavy logging can be really useful when you do not know what's happening. You should activate all warnings and log them to a suitable file (ERW tries to never use a variable that it is unset, so that you can easily track typos in variable names). It is not a good idea to output errors on the page being generated, as they will disturb the output more than necessary. A suitable setup, for instance, is

```
error_reporting = E_ALL
display_errors = Off
display_startup_errors = Off
log_errors = On
error_log = /var/log/httpd/php_log
```

Note that the log file must be writable by the user running `Apache`.

Large uploads

If you plan to use filesets, and foresee uploads of large documents, you should probably set the suitable PHP variables to suit your needs, for example:

```
memory_limit = 20M
post_max_size = 20M
upload_max_filesize = 20M
```

If you plain playing with filesets, it is probably also a good idea to override the default `upload_tmp_dir`.

Configuring HTMLArea

HTMLArea[1] support is a bit experimental, but it works. Essentially, you can use an ERW type (ha) to specify that an area of text should get a WYSIWYG editor. Note that presently HTMLArea works only on Gecko-based browsers and Explorer—in all other supported browsers the effect will be the same as that of a.

First of all, you must download and install HTMLArea 3 (or higher) and set a suitable configuration variable. Then, there are several ways to customise HTMLArea. For the time being, configuration is common to all instances, and contained in the file `js/htmlAreaConfig.js` in the document root. The code contained therein is evaluated immediately after loading HTMLArea (technically, before calling `HTMLArea.init()`) and must assign to the variable `htmlAreaConfig` a function returning an HTMLArea configuration object. Finally, the configuration object can contain an optional `erwPlugins` field, which must be an array containing the names of the plugins you want to register in each HTMLArea instance (it is your responsibility to load those plugins). For more information, have a look at the HTMLArea documentation.

As an example, the sample file provided with ERW loads some plugins, and then just makes the background white. Note that is essential that plugins are loaded *before* `HTMLArea.init()`, whereas they must be registered *after* the creation of an HTMLArea editor:

```
var erwPlugins = [ "TableOperations", "ContextMenu", "CharacterMap" ]
```

```
for(var i in erwPlugins) HTMLArea.loadPlugin(erwPlugins[i])

htmlAreaConfig = function() {
 var config = new HTMLArea.Config()
 config.pageStyle = 'body { background-color: white }'
 config.erwPlugins = erwPlugins
 return config
}
```

Caution

HTMLArea is fantastic in theory, but it is a very fragile piece of software. For instance, if any of the plugins you are using has no translation for the current language, HTMLArea will not start, without giving any message. Always check that you have translations for all plugins you use in all languages specified in `$_ERW_supportedLocales` configuration variable (we use the first two letters of the locale to select HTMLArea's language). A reasonable trick is to add all missing languages as soft links to the English version (and be sure that your server follows soft links).

Another relevant problem is that, as of this writing, HTMLArea ruthlessly overwrites the unload event handler. As a result, users closing a window after having made some changes will not be warned that their changes have been lost. The only solution is removing manually from HTMLArea's code the guiltly lines (search for "unload" in `htmlarea.js`). Moreover, HTMLArea does not call the change event handler when the text changes. As a result, changes performed in an HTMLArea text editor will not cause warning messages about changes being cancelled or lost.

Configuring FCKeditor

FCKeditor[2] is a more supported alternative to HTMLArea. Also FCKeditor works only on Gecko-based browsers and Explorer.

First of all, you must download and install FCKeditor 2 (or higher) and set a suitable configuration variable. Then, FCKeditor can be easily configured using XML and some JavaScript files.

Caution

The warnings given for HTMLArea are valid also here. In particular, at the time of this writing FCKeditor has no officially supported way to set its content after it has been created, so ERW must dig into its internals.

Configuring JSCalendar

JSCalendar[3] can be used to add to date input fields a button that let that user choose a date using a pop-up calendar. For the time being configurability is limited. Essentially, you can use an ERW type (jd) to specify that a JSCalendar button should be added to a date attribute (if JSCalendar is not configured, you will not get the button). JSCalendar should work on all browsers supported by ERW

First of all, you must download and install JSCalendar (or higher) and set a suitable configuration variable. Then, there are several ways to customise JSCalendar. For the time being, you can only configure the look of the calendar by setting a suitable style sheet to be loaded. The default ERW style sheet shows how to load a style sheet.

Caution

The warnings given for HTMLArea are valid also here. Be very careful, in particular, with translations, which are selected automatically.

Notes

1. http://www.htmlarea.com/
2. http://www.fckeditor.net/
3. http://www.dynarch.com/projects/calendar/

Chapter 8. `ERtool`

The first component of ERW you will have to deal with is `ERtool`, which is responsible for analysing ERL files, checking the consistency of the entity-relationship schemata they describe, and produce various outputs depending on the options you specified.

`ERtool` is a Java™ (version 1.4 or greater) application, and as such comes in the form of a *Java archive*, a file containing a number of Java classes in compressed form. The Java archive of ERW is, not surprisingly, named `erw.jar`. When you want to recall `ERtool`, you must be sure that the archive is installed in the extension directory, or that you provide suitable options to Java, or that you set the $CLASSPATH environment variable, so that it can find the class.

Said that, `ERtool` is a Java class whose complete path is `it.unimi.dsi.erw.ERtool`. Thus, you can invoke it with **`java it.unimi.dsi.erw.ERtool <`*`filename`*`>`**, since it expects to read an ERL file from standard input. The archive `erw.jar` contains also a Java *manifest* that allows you to run `ERtool` just with **`java -jar erw.jar <`*`filename`*`>`**.

As a commodity, a command named **ertool** (a script shell that invokes the Java class) is also provided. It takes the same arguments as the Java class.

After searching for the ERL DTD in the current directory, `ERtool` looks in `/usr/share/ERW/dtd` and finally at the directory defined by the Java property `it.unimi.dsi.erw.dtddir`; this property is automatically set by the **ertool** command using the content of the ERW_DTD_DIR environment variable. Indeed, setting this variable is the easiest way to guarantee that the ERL DTD will be available. You can also set the variable manually, using, for instance, **`java -jar erw.jar -Dit.unimi.dsi.erw.dtddir=`*`directory`* `<`*`filename`*`>`**.

> **Warning**
>
> A common problem with running `ERtool` is that it cannot find the ERL Document Type Definition (DTD). Often setting up an SGML/XML system is not so easy, so the simplest thing to do is to put everything in `/usr/share/ERW/dtd`, or to be sure that the current directory contains all DTD-related files when running `ERtool` (you can copy them or create symbolic links), or set up correctly ERW_DTD_DIR.

Options

Without options, `ERtool` will simply read an ERL file from standard input and check it for syntactic and semantic errors. However, there are a number of options that will select a back-end and produce useful output:

`--sql`

Activates the SQL back-end. The entity-relationship schema is reified, and complete SQL code creating all necessary tables is printed to standard output. This code can be saved or even directly piped into a database client.

`--defs`

Activates the definition back-end. A number of PHP files will be created in the current directory, one for each entity type, relationship type, and for each instance of fileset or dynamic enumerative type. These files should be moved in a place accessible to the run-time environment.

`--defs-utf-8`

As above, but instead of using the standard PHP encoding (ISO-8859-1), definition files will be output in UTF-8 encoding. This is essential if you are using the UTF-8 support.

`--forms form1 form2 ...`

Activates the forms back-end. For each form specified (an ERF file), a corresponding PHP file coding the form will be produced. These files should be moved in a place accessible to the run-time environment.

`--forms-utf-8`

As above, but instead of using the standard PHP encoding (ISO-8859-1), forms will be output in UTF-8 encoding. This is essential if you are using the UTF-8 support.

`--default-forms`

Outputs default forms (embedded and non-embedded) for all types in the schema. Compiling those forms and installing the resulting PHP files would not modify ERW behaviour, as they are equivalent to the run-time environment generated default form. However, you can modify them to produce custom forms.

`--doc`

Activates the documentation back-end. A DocBook 4.1.2[1] XML file commenting in detail the entity-relationship schema and the reification process will be printed on standard output. You can then easily convert it to your preferred format.

`--dot`

Activates the DOT language back-end. The output can be formatted using the **dot** utility from the GraphViz[2] suite of tools. More details can be found in the Section called *The DOT graphical back-end*.

`--i18n type`

Activates the internationalisation back-end. The output is a gettext template file that contains all the attributes and leg labels (if `type` is `label`), all the labels of static enumerative types (if `type` is `enum`) or all the help strings (if `type` is `help`). The file can be then used for translations or manipulated with standard gettext commands. See Chapter 11.

Checks

`ERtool` performs a number of checks to ensure that your entity-relationship schema is sensible, that it respect the limitations of the run-time environment, and that it does not contains cardinality constraints so strict that they would make modifications to the database impossible. Most of the error messages are self explanatory, but some explanation is necessary about cardinality constraints.

Recall that an entity-relationship schema is a directed graph. Thus, it makes sense to speak of *directed cycles*, sequences of contiguous arcs (with the right direction) that form a cycle.

The first check performed by `ERtool` is that no such cycle is formed by total injective relationship types. In such a case, it is impossible to modify the database using only local modifications (that is, using ERW), as all entity sets along the cycle are forced to have the same cardinality. In this case `ERtool` will stop with an error.

Then, ERtool checks that there is no cycle formed by total relationship types. To insert an entity in a type belonging to the cycle you need to relate it with some other entity, and this initially impossible. However, if you manually insert some data in the entities, the problem is solved, so ERtool will just warn you.

Finally, it is necessary to make sure that it is impossible that an instance of the schema contains two distinct ownership paths between two entities (in particular, it must not be possible that an entity owns itself); to this purpose, ERW uses a novel algorithm that is able to discover statically if such paths can exist, looking at the type structure of the schema.

Error Messages

Most error message of ERtool are self explanatory. We gather here the most obscure ones.

A relationship type must be multi in both legs or none

> As explained in the Section called *Entity-Relationship Schemata* in Chapter 1, cardinality constraints have some restrictions: if you ask for a multirelation on one leg, you must do the same on the other.

You cannot declare bijections--if both legs have upper 1, one leg must have lower 0

> Bijections are of no use in ERW, because they make entities impossible to create or delete (as the entity sets adjacent to bijections must always have the same number of elements).

You cannot declare injective relations that are not partial functions--they must be transposed

> An injective relation is the transpose of a partial function. For simplicity, in ERW partial functions are always absorbed into their source: if you declare an injective relation that is *not* a partial function, ERtool should absorb it into its target. You just have to exchange the two legs.

Only the first leg can be a key

> For simplicity, in ERW identification functions start from the weak entity type and end on the owner. You just have to exchange the two legs.

The DOT graphical back-end

There is currently no graphical editing tool for ERL files. However, there is a graphical back-end outputting (parts of) a schema in the DOT language. The DOT file can then be processed using standard tools from the GraphViz[3] suite of tools, which contains layout generators, GUI editors, etc.

To activate the back-end you must use the `--dot` option, followed by a (possibly empty) sequence of entity-type identifiers; each of them can be prefixed with `-`. If the sequence is empty, the whole schema is output. Otherwise, if there are just identifiers prefixed with `-`, those types are removed from the output. In the other cases, the set of entity types included in the output is built as follows: every identifier without the `-` prefix adds to the output the type itself, all its supertypes, all the types it owns, and all the types adjacent to the previous ones; every identifier with the `-` prefix removes the corresponding entity type from the output. Each rule is applied following the order of the parameters on the command line.

This mechanism allows you to print modularly parts of a huge schema: for instance, using **ertool --dot -usr -grp -type** you can omit the part of the schema handling au-

thorisation. On the contrary, in a large schema you would include the types you are interested in.

A useful feature of DOT (supported by ERtool) is the possibility of associating a URI to a node. ERtool associates to each entity type and relationship type the relative page of the automatically generated documentation. In other words, if everything is set up correctly, you can look with a PDF viewer at your schema, and clicking on a entity or relationship type you can immediately see all the relevant information. If you prefer, you can even generate an image (say, in PNG format) and a client-side image map that will obtain the same effect in a browser.

Making this work is a bit tedious, because the links are local, and thus must be absolute (usually your browser is already running somewhere, so we cannot rely on its current directory being the one we need). However, they are generated relatively to the current directory, so when generating a DOT file you *must* be in the same directory where you translated the DocBook documentation to HTML. Moreover, the name of the directory containing the documentation must be equal to the id attribute of the root erl element.

Let us work out completely an example (the example works under most Linux distributions). Starting with library.xml, we first generate its documentation and the DOT file:

```
$ ertool --doc <library.xml >libraryDoc.xml
$ ertool --dot <library.xml >library.dot
```

Then, we produce the HTML documentation. Note that we set explicitly the output directory to match the id attribute of the root erl element:

```
$ db2html libraryDoc.xml -o library
```

Finally, assuming GraphViz has been installed successfully, we generate a graphical display:

```
$ dot -Tps2 <library.dot >library.ps
$ ps2pdf -sPAPERSIZE=a0 library.ps
```

There are several options of **dot** that can be used to tweak the output (you can even prefer the spring-embedder, **neato**, but remember to specify the option -Goverlap=false). In the end, the resulting PDF file will contain hyperlinks to the HTML documentation that makes it easy to browse the schema.

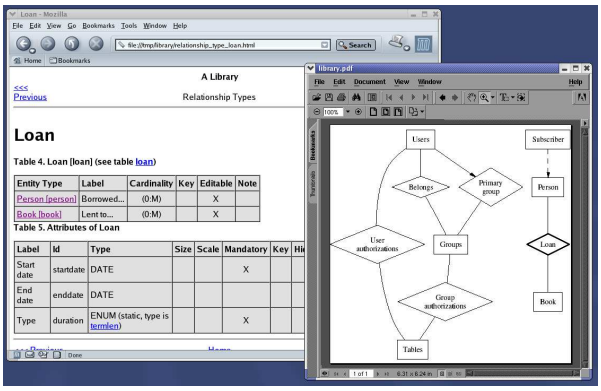

Figure 8-1. An example of automatically laid out entity-relationship diagram with hyperlinked documentation.

The ERW Graphical Notation

The diagrams created by ERtool use graphical hints to suggest cardinality constraints, subtyping and identification functions. The conventions used are, unfortunately, mostly idiosyncratic. After a careful examination of the existing notations, major theoretical or practical flaws where discovered. In some cases, the notation would have worked just for binary relationship types. In other cases, it was utterly counterintuitive.

Thus, ERtool uses its own notation, which is directly derived from Chen's, but also borrows from the standard mathematical notation, Object Role Modelling, and a bit of fantasy.

Entity Types

Entity types are represented by boxes containing the type label. No attributes are shown, as they are hyperlinked, and can thus be easily accessed. The border of abstract types is dashed.

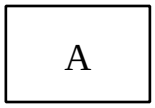

Figure 8-2. An entity type.

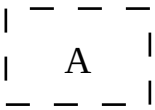

Figure 8-3. An abstract entity type.

Subtyping is represented by a dashed arrow from an entity type to its direct supertype(s).

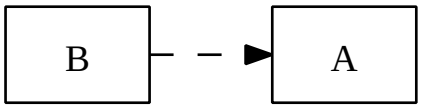

Figure 8-4. An entity type with a subtype.

Weak entities are marked by a double frame, as in standard entity-relationship diagram practise. Identification functions are simply represented by a bold arrow from the weak entity type to the owner (the name is not relevant, as there can be at most one such function between two entity types)

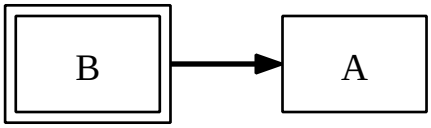

Figure 8-5. An weak entity type and its owner, linked by an identification function.

Relationship Types

Relationship types are represented by diamonds containing the type label (again, no attributes are shown). Relationship types that are to be instantiated to multirelations are shown with a thicker border (as multirelations are somehow fatter than relations).

The representation of the remaining cardinality constraints is very simple: a cardinality constraint of the form `(1:`*x*`)` adds a black dot at the end of the corresponding leg. Thus, when you see a black dot you should read "mandatory" (as in Object Role Modelling). A cardinality constraint of the form `(`*x*`:1)`, instead, generates an arrow tip *near the diamond*. This corresponds to the mathematical fact (explained in detail in the Section called *Cardinality Constraints* in Chapter 1) that such a constraints makes a leg into a partial function from the entity set to the relationship set. Moreover, it shows immediately and intuitively that there is just one way to go from the entity set to the relationship set (and thus to the other component type sets, as legs are functions).

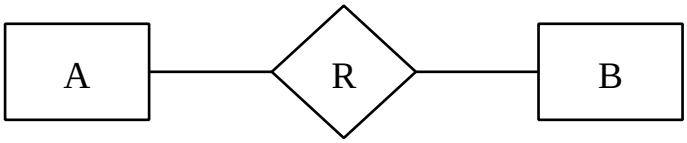

Figure 8-6. `(0:N)/(0:N)`

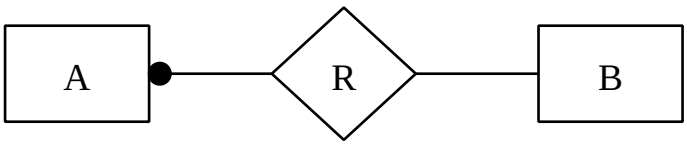

Figure 8-7. `(1:N)/(0:N)`

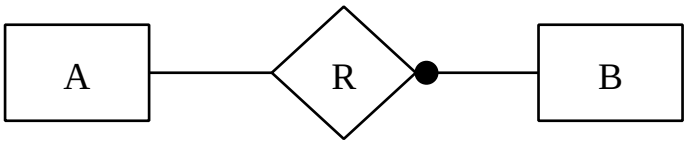

Figure 8-8. `(0:N)/(1:N)`

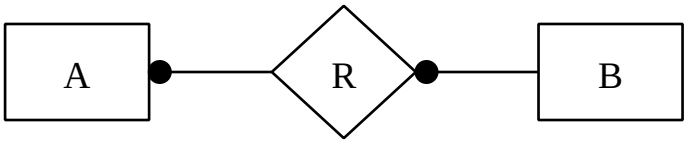

Figure 8-9. (1:N)/(1:N)

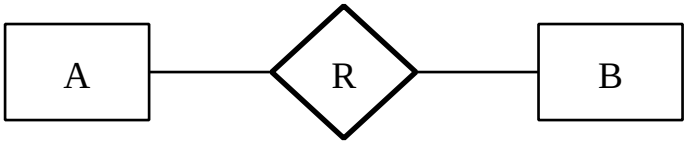

Figure 8-10. (0:M)/(0:M)

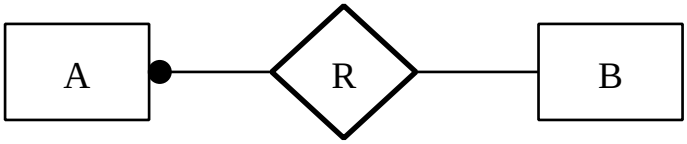

Figure 8-11. (1:M)/(0:M)

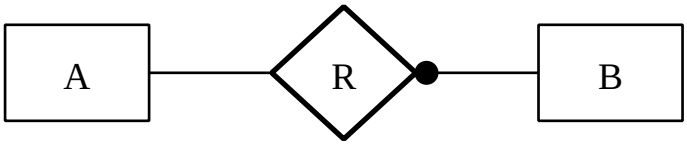

Figure 8-12. (0:M)/(1:M)

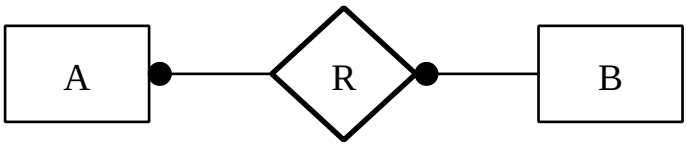

Figure 8-13. (1:M)/(1:M)

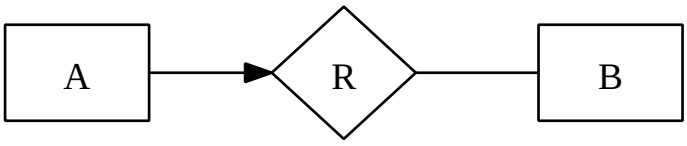

Figure 8-14. (0:1)/(0:N)

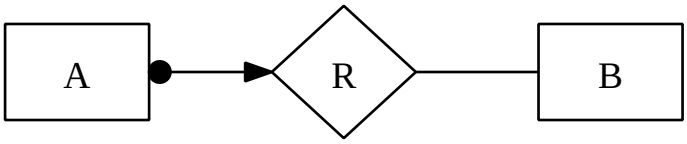

Figure 8-15. (1:1)/(0:N)

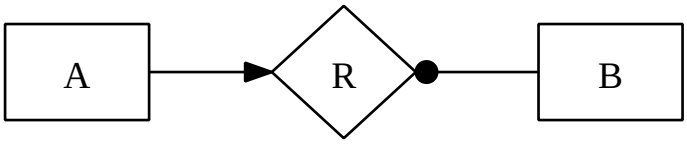

Figure 8-16. (0:1)/(1:N)

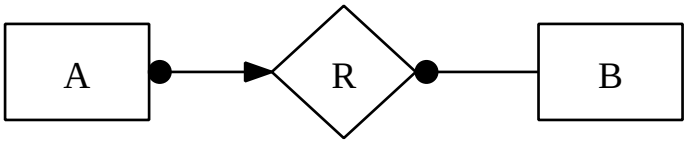

Figure 8-17. (1:1)/(1:N)

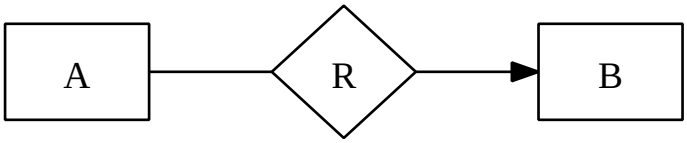

Figure 8-18. (0:N)/(0:1)

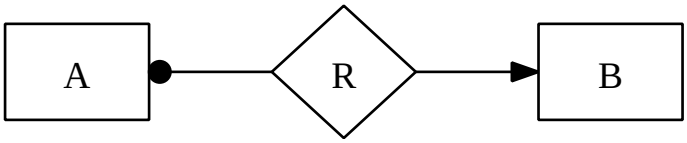

Figure 8-19. (1:N)/(0:1)

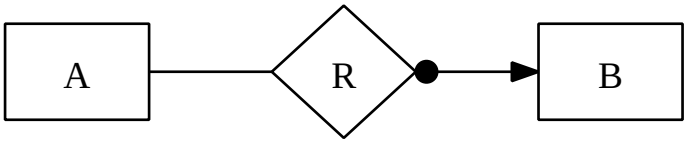

Figure 8-20. (0:N)/(1:1)

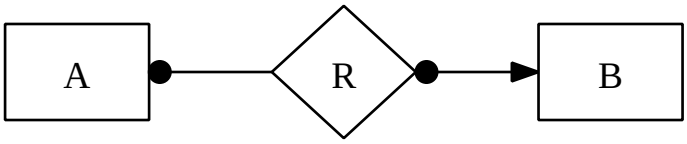

Figure 8-21. `(1:N)/(1:1)`

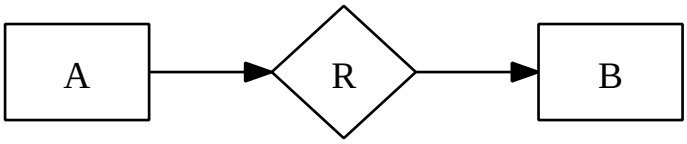

Figure 8-22. `(0:1)/(0:1)`

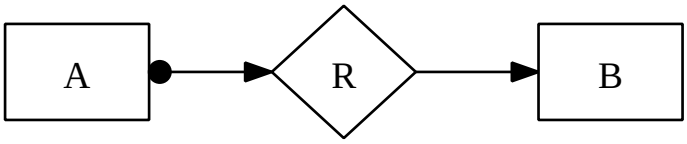

Figure 8-23. `(1:1)/(0:1)`

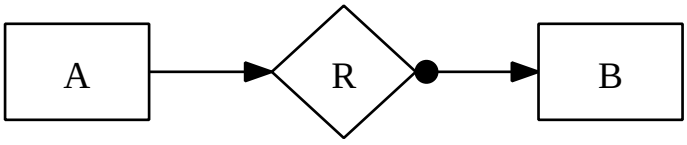

Figure 8-24. `(0:1)/(1:1)`

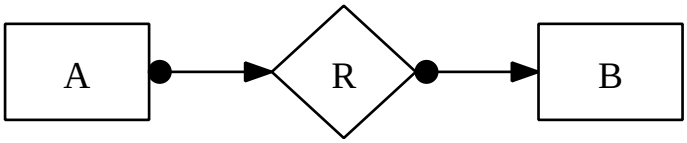

Figure 8-25. `(1:1)/(1:1)`

Notes

1. http://www.oasis-open.org/docbook/
2. http://www.research.att.com/sw/tools/graphviz/
3. http://www.research.att.com/sw/tools/graphviz/

Chapter 9. Standards and Browser Requirements

ERW heavily relies on browser compliance with respect to a number of standards. Presently only Mozilla (and thus Gecko-based browsers, such as Netscape 6 or Galeon), Konqueror (and thus KHTML-based browsers, such as Safari) and MicrosoftÂ® Internet Explorer from version 5.5 up have a sufficiently compliant implementation of these standards to work properly. If ERW detects a browser with insufficient capabilities, it will warn the user.

HTML 4.01

ERW use all features of the W3C standard, including lesser-known elements such as `IFRAME`, `LABEL`, `FIELDSET` and `LEGEND`.

CSS

ERW uses CSS1 and some features of CSS2.

The W3C Document Object Model

ERW access and dynamically updates HTML elements using Level 1 of the W3C Document Object Model specification, plus bits of Level 2 (mainly for style).

ECMAScript (ECMA-262)

ERW uses as a scripting language the standardised version of the JavaScript language originally developed by Netscape.

Chapter 10. Useful Functions

In this chapter we discuss functions defined in the run-time environment that also users writing hooks could find useful.

Database Functions

When executing a hook, you can access some static functions of the `ERW` class that can simplify your access to the database. Note that all these functions *stop the current script and write to the log file* in case of an error.

`ERW::log(`*`file`*`, `*`line`*`, `*`message`*`, [`*`level`*`]);`

Logs a message if the current logging level is equal to or below the one specified in the main configuration file. Possible levels are `ERW_DEBUG`, `ERW_INFO`, `ERW_WARNING` and `ERW_ERROR` (the default). The first two parameters are usually filled using the PHP macros `__FILE__` and `__LINE__`, respectively.

`ERW::quote(`*`string or array`*`);`

Quotes a string for SQL processing. Note that, when applied to an array of strings, it will quote by reference the whole array.

`ERW::unquote(`*`string`*`);`

Undoes what `EWR::quote()` did (but works by value).

`ERW::query(`*`query`*`);`

Executes the given query and returns the result. Database errors cause logging and immediate exit.

`ERW::select(`*`select query`*`);`

It is identical to `ERW::query()`, but it automatically prefixes its argument with "`SELECT `", so you can write things like `ERW::select("* from type");`.

`ERW::getRow(`*`select query`*`);`

It is identical to `ERW::select()`, but it automatically returns the first row of the result set.

```
ERW::getOne(select query);
```

It is identical to `ERW::getRow()`, but it automatically returns the first column of the row. Very useful when you just want a single datum from the database.

JavaScript Functions

To use these function, you must know some convention about the HTML elements identifiers generated by ERW. This is necessary, as to create useful hooks you need to be able to detect which row of a table the user selected.

Each relevant HTML element in a page generated by ERW has a well-defined identifier. In particular, whenever a list of elements is displayed to the user, the corresponding table has an identifier (in the XML sense) generated by prefixing a letter followed by the underscore character to a table name. More precisely:

- main lists are named `m_type`;
- in the editing of relationships, the list that lets the user choose entities is named `s_relationship-type`, whereas the list showing relationships is named `r_relationship-type`;
- lists that lets the user edit a fileset are named `f_entity-type_attribute-name`.

Said that, you can use the following functions to discover which element of a list has been selected by the user:

```
int getSelectedId(elemId);
```

Given the identifier of a list, this function returns the SQL identifier (i.e., the value of the `id` column) of the selected element.

```
int getSelectedItem(elemId);
```

Given the identifier of a list, this function returns the 0-based index of the selected row.

```
void openServiceWindow(url, name);
```

This function opens a new window without toolbar, menus, etc. and loads it with the given URL. The given name will be juxtaposed to the current number of milliseconds to generate a name for the window, so that two activations of the same button actually open two different windows.

You should look at the example given in the Section called *The Button Hook* in Chapter 4.

Chapter 11. Localisation

All ERW strings can be easily localised using the GNU gettext package. It is a fairly standard way of giving translations of literal strings in various languages, and you can find documentation about it at the GNU site[1]. Moreover, there is limited support for some locale-dependent quirks such as the decimal point.

There is also support for UTF-8, using the PHP `mbstring` extension.

Language Configuration Variables

Locales (country-specific languages) are specified in ERW as in RFC 2616 for the `Accept-Language` HTTP header, that is, a two-letter language code, a hyphen and a two-letter country code. Thus, `it-IT` is the Italian spoken in Italy, whereas `en-UK` is British English.

If you want to support some languages, you must specify them in the `$_ERW_supportedLocales` configuration variable. Then, for each language, you have to provide a file that, for each string used in ERW, provides a translation. An example of such a file for Italian is given in the file `ERW.it.po` that comes with the distribution. You should write a similar file for your language (and possibly send it to me[2]), compile it using the **msgfmt** utility coming with gettext, and finally installing the resulting `ERW.mo` file in the proper directory (e.g., `it/LC_MESSAGES` for Italian) inside the path specified by the `$_ERW_localePath` configuration variable. Then, ERW will fetch from the browser the preferred language and try to match it with one of those supported. You can also set a default locale (instead of `en-US`) or a fixed locale (see Chapter 7).

The translation file above accounts for all strings built into ERW. However, you may want to translate other parts of the user interface, such as labels, enumerative types and help attributes. This is possible simply by creating translation files named `ERW-`*`id`*`-labels`, `ERW-`*`id`*`-displays`, `ERW-`*`id`*`-enums` and `ERW-`*`id`*`-help`, where *`id`* is the identifier of the root element of your ERL file (e.g., "library" in our main example). These files should contain translations for all labels, display customisation strings, enumerative-type labels and help text (i.e., content of `help` attributes), respectively.

The translation files are usually derived from templates produced by `ERtool` (as it happens with templates produced by **xgettext**). Note that gettext offers several utilities that makes translations much easier (automatic compendia, comparison of translation and template files, and so on).

Other Localisation Variables

As already mentioned in the Section called *The `ERW-conf.php` Initialisation File* in Chapter 7, you have some configuration variables that let you decide which character to use as decimal point and decimal comma. Moreover, you can choose between three date formats, namely ISO (year/month/day, very sensible), European (day/month/year, sensible), and US (month/day/year, broken).

UTF-8 Support

Natively, ERW runs using ISO-8859-1. This is the standard character encoding for most PHP installation, and for several databases. However, using ISO-8859-1 is a strong limiting factor if you plan to manage data in languages that are not covered by its codepoints.

ERW supports fully UTF-8. As usual when character-encoding problems are involved, the mechanism is not so simple. The first step to use UTF-8 support is to make PHP UTF-8 aware. This requires compiling in PHP the `mbstring` extension,

which allows to handle transparently strings in several encodings. Unfortunately, several distributions do *not* release PHP with the `mbstring` extension enabled, so you may have to recompile PHP (although if you have a decent packaging system this should turn out to be rather easy). To check whether your PHP installation has `mbstring`, you can look at the output of **php -m**.

Once the `mbstring` extension has been loaded, you must enable UTF-8 support by setting the configuration variable `$_ERW_utf8` and configure PHP to use UTF-8 by modifying the following parameter in the PHP configuration file:

```
mbstring.func_overload = 7
```

It is essential that all functions are overloaded with those of `mbstring`, otherwise ERW will stop and log an error message. Note that this parameter may be set in the main configuration file only.

ERW will generate a suitable HTML tag that will inform the browser that its output is encoded as UTF-8. However, it is good practise to communicate this information *before* the page is actually output, using an HTTP header; this can be obtained setting the following PHP configuration variable:

```
default_charset = "UTF-8"
```

The value of `default_charset` can also be changed locally, using an `.htaccess` file. Be sure to have the required permissions (`AllowOverride Options`), and just add a line

```
php_value default_charset UTF-8
```

The rationale behind UTF-8 support is that *everything should be in UTF-8*. Thus, with UTF-8 support activated web pages are output in UTF-8, input from the browser is read in UTF-8, and data exchange with the database if performed using UTF-8 strings. This influences a number of settings, going from the DBMS internal encoding to the terminal encoding, that must be set up correctly.

Note that usually an 8-bit clean database with work flawlessly with UTF-8 support, as UTF-8 has several good properties (e.g., lexicographical byte-by-byte ordering coincide with lexicographical character-by-character ordering).

Warning

By default, `ERtool` will generated definition files using ISO-8859-1. As long as none of your labels, enumerative types, etc. is not US-ASCII, this is not a problem. If, however, you plan to use arbitrary UTF-8 characters in your labels, you should use the UTF-8 backend when producing definition files with `ERtool`.

Notes

1. http://www.gnu.org/
2. mailto:vigna@acm.org

Chapter 12. Script Reference

This chapter gives an overview of the purpose of some scripts coming with ERW. Its main purpose is to make you able to create a custom portal for access to the ERW editing and customisation pages.

ERW/checkERW.php

This script is never called during editing. It checks the structure of definition files and custom files, looking for mistakes in array keys, parameter types and so on. It is a fundamental tool, as it is easy to make a typo. For a correct setup it should give no message.

There are two ways to run this script: either you create a symbolic link to it in your document root, and recall it as a URL (e.g., `http://example.com/checkERW.php`), or you start it from the command line using **`php -C -f ERW/checkERW.php`**in the latter case, your current directory must be your document root. Note that if you installed the `ERW` directory in a special place, you will have to give the complete path, for instance, **`php -C -f /usr/share/php/ERW/checkERW.php`**

> **Note:** It may happen that your custom files contain some special code that sets some customisations conditionally. Usually this code will not work when run out of context by `checkERW.php`. However, `checkERW.php` sets a variable named `$_ERW_check`: by testing whether the variable exists, you can discover whether you are being run by `checkERW.php` and skip problematic code.

ERW/checkdb.php

This script is never called during editing. It checks the overall database integrity, including referential integrity, logical integrity, correspondence between ERW definition files and SQL table definitions, and so on. For a correct system it should give no message. The methods to invoke this script are the same of checkERW.php.

> **Warning**
>
> Even for a small database, the running time of this script will certainly exceed the default maximum execution time for a script set in the PHP main configuration file. This should not be a problem, because the script tries to set for itself unlimited execution time. However, if you set safe mode on in the PHP main configuration file, the setting will have no effect.

ERW.php

This script contains all of the common code.

list.php

This script creates a *main list*, that is, a list of entities of a given type, and provides buttons to modify them. Its only parameter is `type`, the name of an entity, relationship or dynamic enumerative type.

form.php

This script creates a *form* to edit an entity. Its parameters are `type`, the name of an entity, relationship or dynamic enumerative type, and `id`, which provides the SQL identifier of the element to edit (if it is zero, a new element will be created).

default.php

This script generates the CSS style sheet used by all other scripts. It can be tweaked to suit your needs.

chown.php

This script allows to change the owner and the group of an element. It has the same parameters as form.php.

setprefs.php

This script allows the user to change some preferences, including font size and the number of columns/rows of editing elements. It has no parameters, as it will simply try to set a cookie.

Chapter 13. Troubleshooting

As for any web application, troubleshooting a ERW installation is not easy. Many components must collaborate correctly (HTTP server, DBMS, etc.) to get everything working, and diagnosing which component caused the failure is often difficult. Some of the most sophisticated scripts, for instance, are loaded in hidden frames and seeing their output is, at best, difficult. Here we suggest a number of checkpoints to troubleshoot problems, starting from database reification up to user interface customisations. We assume that you have written a valid ERL file—you can tell this by running `ERtool` without options.

`ERtool` problems

1. The most common problem with `ERtool` is that it cannot find the ERL DTD. If you did not insert all DTD-related files into your SGML/XML catalogues, you must be sure that *all DTD-related files are located in the current directory* whenever you use `ERtool` (also symbolic links will do the job).
2. If Java complains that it is not finding some classes, check that the `ERtool` Java archive is properly installed, that you have Java 1.4 or greater, and that you have an XML parser (this should be automatically true with version 1.4).

PHP problems

1. `Warning: The gettext PHP extension has not been loaded`. This error message in the logs means that your PHP installation does not include the gettext extensions. If you want to use localisation, you must reconfigure properly PHP

SQL problems

1. First of all, check that your database does not report any error when fed with the SQL output of `ERtool`.
2. If errors are reported, check that they are not due to custom default function specified by `default` attributes. Strip all default attributes from your ERL file and try again.
3. If errors persist, check the same file with at least another DBMS; it can happen that some database does not support the fragment of SQL-99 used by ERW.

Configuration and Customisation Problems

1. First of all, check that both `checkERW.php` and `checkdb.php` do not report any error about your ERW configuration and database content. In particular, if you upgraded from a version without original filenames, a missing column `original` will be reported in each fileset. Please regenerate the SQL code to see how these columns should be created.
2. If you have custom files, remove them gradually, so that you can tell whether the problem is caused by a customisation or by a hook.

When Nothing Seems to Work

1. First of all, check that you have the very last version of all components involved. This means a recent version of the DBMS, of the HTTP server, of PHP, of PEAR, and of the browser.
2. Force your browser to reload all files. Even if you have the last version of ERW, your browser could be caching old scripts. If you can, erase the cache altogether.
3. Then, *check the logs*. You should have the maximum level of logging enabled in your PHP configuration file (including logging of all warnings, i.e., `error_reporting = E_ALL`) and also the ERW logging set to the maximum level (i.e., `$_ERW_logLevel=ERW_DEBUG;`). Warnings about usage of undefined variables usually show that you have misdefined some customisation variables or hooks. In this case, try first to disable all custom hooks and scripts.
4. If everything seems to be fine from the PHP side, check with Mozilla and its JavaScript console that no JavaScript errors are generated. Please *try with the last Mozilla version* before doing any bug reporting.

Reporting Bugs

If you really think you hit a bug, please report it. It is important that you provide

- the ERL file that generates your database;
- any custom file that are necessary to replicate the bug (but *check carefully* that the problem does not lie in the custom files!);
- PHP, HTTP server and JavaScript logging.

Without these items, it is virtually impossible to diagnose a problem.

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

Copyright Â© 2000 Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A

copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the rÃ´le of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties -- for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This Licence

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Notes

1. http://www.gnu.org/copyleft/